XML Prague 2019

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 7–9, 2019

XML Prague 2019 – Conference Proceedings
Copyright © 2019 Jiří Kosek

ISBN 978-80-906259-6-9 (pdf)
ISBN 978-80-906259-7-6 (ePub)

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 vii

Sponsors ix

Preface xi

Task Abstraction for XPath Derived Languages –
Debbie Lockett and Adam Retter 1

A novel approach to XSLT-based Schematron validation – *David Maus* 57

Authoring DSLs in Spreadsheets Using XML Technologies – *Alan Painter* 67

How to configure an editor – *Martin Middel* 103

Discover the Power of SQF – *Octavian Nadolu and Nico Kutscherauer* 117

Tagdiff: a diffing tool for highlighting differences in text-oriented XML –
Cyril Briquet 143

Merge and Graft: Two Twins That Need To Grow Apart –
Robin La Fontaine and Nigel Whitaker 163

The Design and Implementation of FusionDB – *Adam Retter* 179

xqerl_db: Database Layer in xqerl – *Zachary N. Dean* 215

An XSLT compiler written in XSLT: can it perform? –
Michael Kay and John Lumley 223

XProc in XSLT: Why and Why Not – *Liam Quin* 255

Merging The Swedish Code of Statutes (SFS) – *Ari Nordström* 265

JLIFF, Creating a JSON Serialization of OASIS XLIFF –
David Filip, Phil Ritchie, and Robert van Engelen 295

History and the Future of Markup – *Michael Piotrowski* 323

Splitting XML Documents at Milestone Elements – *Gerrit Imsieke* 335

Sonar XSL – *Jim Etevenard* 355

Copy-fitting for Fun and Profit – *Tony Graham* 363

RDFe – expression-based mapping of XML documents to RDF triples –
Hans-Juergen Rennau 381

Trialling a new JATS-XML workflow for scientific publishing – *Tamir Hassan* . 405
On the Specification of Invisible XML – *Steven Pemberton* 413

General Information

Date

February 7th, 8th and 9th, 2019

Location

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

Organizing Committee

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

Program Committee

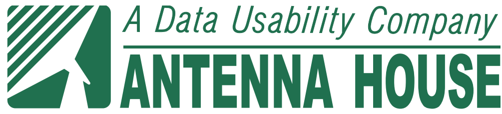
Robin Berjon, *The New York Times*
Petr Cimprich, *Wunderman*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Ari Nordström, *Karnov Group*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Bloomsbury Publishing plc*
Felix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



Preface

This publication contains papers presented during the XML Prague 2019 conference.

In its 14th 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 7–9 February 2019 at the campus of University of Economics in Prague. XML Prague 2019 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, University of Economics in Prague.

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

The Thursday runs in an 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. Additionally, we coordinate, support and provide space for XProc working group meeting collocated with XML Prague.

We hope that you enjoy XML Prague 2019!

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

Task Abstraction for XPath Derived Languages

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Adam Retter
Evolved Binary
<adam@evolvedbinary.com>

Abstract

XPDLs (XPath Derived Languages) such as XQuery and XSLT have been pushed beyond the envisaged scope of their designers. Perversions such as processing Binary Streams, File System Navigation, and Asynchronous Browser DOM Mutation have all been witnessed.

Many of these novel applications of XPDLs intentionally incorporate non-sequential and/or concurrent evaluation and embrace side effects to achieve their purpose.

To arrive at a solution for safely managing side effects and concurrent execution, this paper first surveys both the available XPDL vendor extensions and approaches offered in non-XPDLs, and then describes EXPath Tasks, a novel solution derived for the safe evaluation of side effects in XPDLs which respects both sequential and concurrent execution.

1. Introduction

XPath 1.0 was originally designed to “provide a common syntax and semantics for functionality shared between XSL Transformations and XPointer” [1], and XPath 2.0 pushed the abstraction further by declaring “XPath is designed to be embedded in a host language such as XSL Transformations ... or XQuery” [2]. For XML processing, XPath has enjoyed an arguably unparalleled level of language adoption through reuse, forming the basis of XPointer, XSLT, XQuery, XForms, XProc, Schematron, JSONiq, and others. XPath has also had a wide influence outside of XML, with concepts and syntax being reused in other languages like AQL (Arrango Query Language), Cypher, JSONPath, and OData (Open Data Protocol) amongst others.

As functional languages, XPDLs such as XQuery were designed to avoid strict or ordered evaluation [21], thus leaving them open to optimisations which may exploit concurrency or parallelism. XPDLs are thus good candidates for event

driven and task based concurrent and/or parallel processing. Since 2001, when the first non-embedded multi-core processor the IBM Power 4 [11] was introduced, CPU manufacturers have followed the trend of offering improved performance through greater numbers of parallel hardware threads as opposed to increased clock speeds. Unfortunately, exploiting the performance of additional hardware threads puts an additional burden on developers by requiring the use of low-level complex concurrent programming techniques [12]. Such low-level concurrent programming is often error-prone [13], so it is desirable to employ higher level abstractions such as event driven architectures [14], or task based computation with Futures [16] and Promises [18]. This paper advances the use of XPDLs in this context.

Indeed, the formal semantics for XPath state that "[XPath/XQuery] is a functional language" [4]. From this we can infer that strict XPDLs must therefore also be functional languages; this inference is strengthened by XQuery and XSLT which are both functional languages. By placing restrictions on expression formulation, composition, and evaluation, functional programming languages can enable advantageous classes of verification and optimisation when compared to imperative languages.

One such restriction enforced by functional languages is the elimination of *side effects*. A side effect is defined as a function or expression modifying some state which is external to its local environment, this includes:

1. Modifying either: a global variable, static local variable, or variable passed by reference.
2. Performing I/O.
3. Calling other side effect functions.

XPath and the XPDLs as defined by the W3C specifications address these concerns and prevent side effects by enforcing that:

1. Global variables and static local variables are immutable, and that variables are always passed by value and not reference.
2. I/O is frozen before evaluation, only documents known to the immutable static context may be read, whilst the only output facility is the XDM result of the program.
3. There are no side-effecting functions[1].

In reality though many XPDL implementations offer additional vendor-specific "*extensions*" which compromise functional integrity to permit side effects so that

[1]XPath 3.0 defines only one absolute non-deterministic function `fn:error`, and several other functions (`fn:analyze-string`, `fn:parse-xml`, `fn:parse-xml-fragment`, `fn:json-to-xml`, and `fn:transform`) which could be non-deterministic depending on implementation choices. We devalue the significance of `fn:error`'s side effect by tendering that, it could equally have been specified as a language expression for raising exceptions as opposed to a function.

I/O can be more easily achieved by the developer. Of concern for this paper is the ability to utilize XPDLs for complex I/O requiring side effects without compromising functional integrity or correctness of the application.

The key contributions of this paper are:

1. A survey of XPDL vendor implementations, detailing both how they manage side effects and any proprietary extensions they offer for concurrent execution. See Section 2.
2. A survey of currently popular mechanisms for concurrent programming in non-XPDLs, their ability to manage side effects, and their potential for XPDLs. See Section 3
3. EXPath Tasks, a module of XPath functions defined for managing computational side effects and enabling concurrent and asynchronous programming. To demonstrate the applicability of EXPath Tasks, we offer experimental reference implementations of this module in XQuery, XSLT, Java (for use from XQuery in eXist-db), and JavaScript (for use from XSLT in Saxon-JS). See Section 4.

We next briefly examine the original vision for XPath, XQuery, and XSLT, with particular concern for how these languages should be evaluated by processors. We then examine how the use of these languages has evolved over time and some of the unexpected and novel ways in which they have been used.

1.1. The vision of XPDLs

The design requirements of XPath 2.0 [3] mostly focused on that of exploiting the XDM (XQuery and XPath Data Model) and interoperability. As a language designed to describe the processing abstractions of various host languages, it did not need to state how the evaluation of such abstractions should take place, although we find that it was not without sympathy for implementations, as one of the stated Goals was: “Enable improved processor efficiency”; unfortunately, we found little explicit public information on how or if that goal was met.

Examining the XQuery 1.0 requirements [6] we find a similar focus upon the XDM, where querying different types of XML documents, and non-XML data sources is possible, provided that both can present their data in an XDM form. However, the XQuery 1.0 specification makes an explicit statement about evaluation: “an implementation is free to use any strategy or algorithm whose result conforms to the specifications in this document”, thus giving implementations a great deal of freedom in how the query should be evaluated.

One of the requirements of XSLT 2.0 is labelled “2.11 Could Improve Efficiency of Transformations on Large Documents” [5]. It describes both the situation where the tree representation of source documents may exceed memory requirements, and a desire to still be able to process such large documents. It uses

non-prescriptive language to suggest two possible solutions: 1) a subset of the language which would not require random access to the source tree, we could likely recognise XSLT 3.0 Streaming as the implementation of that solution, and 2) splitting a tree into sub-trees, performing a transformation on each sub-tree, and then copying the results to the final result tree. Whilst XSLT 2.0 does not state how an implementation should be achieved, many would likely recognise that (2) is an *embarrassingly parallel* problem that would likely benefit from a MapReduce [19] like approach.

An academic example of exploiting the implicit parallelisation opportunites of XPDLs is PAXQuery, which compiles a subset of XQuery down into MapReduce jobs which can execute in a highly-parallel manner over a cluster of Hadoop nodes [24]. To the best of our knowledge, Saxon is the only commercial XPDL processor which attempts implicit parallelisation. However, Michael Kay reports that within the XSLT processor it can be difficult to determine when implicitly parallelising operations will reduce processing time [20]. Saxon therefore also offers vendor extensions which allow an XSLT developer with a holistic view of both the XSLT and the data it must process, to explicitly annotate certain XSLT instructions as parallelisable.

1.2. Novel applications of XPDLs

XPDLs have been used in many novel situations for which they were never envisaged, many of which utilise non-standardised extensions for I/O side effects and concurrent processing to achieve their goals.

1.2.1. XPDLs as Web Languages

XPDLs, in particular XQuery, have been adopted with considerable success as server-side scripting languages for the creation of dynamic web pages and web APIs. A web page is by definition a document, and since an HTML document is representable as an XML document, XPDLs' ability to build and transform such documents from constituent parts has contributed to their uptake. Implementations such as BaseX, eXist-db, and MarkLogic all provide HTTP Servers which execute XQuery in response to HTTP requests. Whilst a single XQuery may be executed concurrently by many thousands of users in response to incoming HTTP requests, stateful information often needs to be persisted and shared on the server. This could be in response to either a user logging into a secure website, at which point the server must establish a session for the user and memorize the users identity; or multiple web users communicating through the server, for example, updating stock inventory for a shopping basket or social messaging. Regardless, such operations require the XPDL to make side-effecting changes to the state of the server or related systems.

XSLT's main strength as a transformation language for XML is equally applicable to constructing or styling HTML web pages. Web browsers offer limited XSLT 1.0 facilities, which can be either applied to XML documents which include an appropriate Processing Instruction, or invoked from JavaScript. The XSLT process offered by the web browser vendors is a black-box transformation, whereby an XSLT stylesheet is applied to an XML input document, which produces output. This XSLT process is completely isolated and has no knowledge of the environment from which it is called; it can not read or write directly to or from the web page displayed by the browser. In contrast, in recent years Saxonica has provided JavaScript based processors which run directly within the web browser, removing the isolation and allowing access to the web page state and events via XSLT extensions. First with Saxon-CE, a ported version of the XSLT 2.0 Saxon Java processor, and then with Saxon-JS, a clean implementation of XSLT 3.0 in JavaScript. The XSLT extensions designed for use with these processors make use of asynchronous processing (as demanded by JavaScript) and side effects to read and write the DOM model of the web page.

Similar to Saxon-CE, although now unmaintained another notable example is XQiB. XQiB implements an XQuery 1.0 processor in JavaScript which runs in the web browser and provides a number of XQuery extension functions which cause side effects by writing directly to the HTML DOM and CSS [25].

1.2.2. Binary Processing with XPDLs

The generation of various binary formats using XPDLs has also been demonstrated. One such example is Philip Fennel's generation of TIFF format images, which uses a Reyes pipeline written in XSLT [26]. One of Fennel's conclusions with regard to execution was that “Certainly it is not fast and it is not very efficient either”. It is not hard to imagine that if concurrent processing was applied to each stage of the pipeline, so that stages were processed in parallel, then execution time might be significantly reduced.

Two XPath function extension modules produced by the EXPath project, the Binary [27] and File Module [28] specifications, allow the user to both read and write files and manipulate binary data at the byte level from within XPDLs. In particular, the File Module, which provides I/O functions, states that some functions are labelled as *non-deterministic*; this specification lacks the detail required to determine if implementations are forced to produce side effects when the functions are evaluated, or whether they are allowed to operate on a static context and only apply I/O updates after execution. The authors of this paper believe that it would be beneficial to have a more formal model within that specification, possibly one which allows implementers flexibility to determine the scope of side effects.

1.3. Motivation

To enable side effects in a web application running in Saxon-JS, the IXSL (Interactive XSLT) extensions (instructions, functions and modes) are provided (as previously developed for Saxon-CE, with some further additions and improvements). These IXSL extensions allow rich interactive client-side applications to be written directly in XSLT.

Saxon-CE used a Pending Update List (PUL) to make all HTML page DOM updates (side effects) at the end of a transform (e.g. setting attributes on HTML page nodes using `ixsl:set-attribute`; and adding content to the HTML page using `xsl:result-document`.) Currently Saxon-JS does not use a PUL, instead these side-effecting changes are allowed to execute immediately as the instructions are evaluated, and it is up to the developer of a Saxon-JS application to ensure that adverse affects are avoided. Since inception, the intention has been to eventually provide better implicit handling. Should the use of PULs be reinstated, or is there an alternative solution?

Meanwhile, use of asynchronous (concurrent) processing is essential for user-friendly modern web applications. Whenever the client-side needs to interact with the server-side, to retrieve resources, or make other HTTP requests, this should be done asynchronously. The application may provide a "processing, please wait" message to the user, but it should not just stop due to blocking.

The `ixsl:schedule-action` instruction allows the developer to make use of concurrent threads, and in particular allows for asynchronous processing. In Saxon-JS, different attributes are defined to cater for specific cases where there is a known need. The `document` attribute is used to initiate asynchronous document fetches; the `http-request` attribute is used for making general asynchronous HTTP requests; and the `wait` attribute was designed to force a delay (e.g. to enable animation), but actually simply provides a way to start any concurrent process. Effectively this provides a mechanism for forking, but there is no offical joining. Are there cases that require a join? Are there other operations which a developer could want to make asynchronously? Rather than building IXSL extensions for each operation, we would prefer to realise a general mechanism for asynchronous processing in XPDLs and by extension XSLT. Continually updating the syntax and implementation of `ixsl:schedule-action`, each time a new requirement arises (e.g. how to allow HTTP requests to be aborted), is not ideal. In particular, the IXSL HTTP request facility was based on the first EXPath HTTP Client Module, recent work on a second version [23] of that module could be advantageous for us. However, by itself it neither prescribes synchronous or asynchronous operation. So, how could we implement in a manner which is both asynchronous and more abstract, requiring few, if any, changes to add additional modules in future?

1.4. Our Requirements

Applications that cannot perform I/O and/or launch parallel processes are unusual. Both I/O and starting parallel processes are side effects, and as discussed, explicitly forbidden within XPDLs, although often permitted by vendors at the cost of imperative interpretation and lost optimisation opportunities.

We aim to break the trade-off between program correctness and deoptimisation in XPDLs. We require a mechanism that satisfies the following requirements:

- A mechanism for formulating processes which manage side effects, yet at the same time remains within the pure functional approach dictated by the XPDL formal semantics.
- Permits some form of parallel or concurrent operation, that is implementable on systems that offer either preemptive or cooperative multitasking.
- Allows parallelisation to be explicitly described, but that should not limit the opportunities for implicit parallelisation.
- Any parallel operation explicitly initiated by the developer, should be cancellable.
- Composability: it should be possible to explicitly compose many side-effecting processes together in a manner that is both pure and ensures correct order of execution.

Regardless of the mechanism, we require that it should be as widely applicable as possible, therefore it should be either:

- Formulated strictly in terms of XPath constructs so that it be reused by any XPDL.

  Ideally, rather than developing a superset of the XPath grammar, a module of XPath extension functions should be defined. The module approach has been successfully demonstrated by the EXPath project, and would likely lower the barrier to adoption.
- A clearly defined abstract processing model which can have multiple syntactical expressions.

  Such a model could for example provide one function-based syntax for XQuery, and another instruction-based syntax for XSLT.

2. Current Approaches by Implementers

This survey provides a brief review of the offerings of the most visible XQuery and XSLT implementations for both concurrent and/or asynchronous execution, and how they manage side effects.

2.1. BaseX

For concurrent processing from within XQuery, BaseX provides two mechanisms: a Jobs XQuery extension module [8], and an XQuery extension function called `xquery:fork-join`. The latter is actually an adoption of xq-promises's `promise:fork-join` function, which we cover in detail in Section 2.5. The former, the Jobs Module, allows an XQuery to start another XQuery by calling either of two XPath functions `jobs:invoke` or `jobs:eval`. Options can be supplied to both of these functions, which instead of executing the query immediately, schedule it for later execution. Whilst deferred scheduled queries are possibly executed concurrently we will not consider them further here as our focus is concurrent processing for the purposes of completing an immediate task. BaseX describes these functions as *asynchronous*, and whilst technically true, unlike other asynchronous programming models the caller neither provides a callback function nor receives a promise, and instead has to either poll or wait in the main query for the result. We believe these functions could more aptly be described as *non-blocking*.

Asynchronously starting another XQuery in BaseX returns an identifier which can be used to either stop the asynchronously executing query, retrieve its result (if it has completed), or to wait until it has completed. The lifetime of the asynchronously executing query is not dependent on the initiating query, and may continue executing after the main query has completed. In many ways this is very similar to a Future (see Section 3.5).

BaseX implements XQuery Update [7] which allows updates to XML nodes to be described from within XQuery via additional update statement syntax. XQuery Update makes use of a PUL (Pending Update List) which holds a set of Update Primitives. These Update Primitives describe changes that will be made, but have not yet been applied. These changes are not visible to the executing query, the PUL is output alongside the XDM when the query completes. This is not entirely dissimilar to how Haskell eventually evaluates an IO monad (see Section 3.4). To further facilitate additional common tasks required in a document database without conflicting with XQuery Update or resorting to side effects within an executing query, BaseX also provides many vendor specific Update Primitives in addition to those of XQuery Update. These include primitives for database operations to replace, rename and delete documents; manage users; and backup and restore databases [29]. The use of an XQuery Update PUL avoids side effects for updates, as it only describes what will happen at evaluation time, leaving the actual updates to be applied at execution time. Ultimately BaseX applies the PUL to modify the state of its database after the query completes and the transaction is committed, thus making the updates visible to subsequent transactions.

Regardless of its support for PULs, BaseX does not quite manage to entirely avoid side effects during the execution of some queries. BaseX offers a number of

XQuery extension functions which are known to cause side effects, including for example, those of the EXPath HTTP and File Modules. Internally such side-effecting functions are annotated as *nondeterministic*, and will be treated differently by BaseX's query compiler. By skipping a range of otherwise possible query optimisations, BaseX ensures that the execution order of the functions within a query is as a user would expect even when these nondeterministic functions are present. In the presence of nondeterminism, optimisations that are skipped include: pre-evaluation, reordering of let clauses, variable inlining, and disposal of expressions that yield an empty sequence.

2.2. eXist-db

eXist-db does not present a cohesive solution for concurrent processing from within XQuery. Until recently, eXist-db had a non-blocking XPath extension function named `util:eval-async` [9] which could start another XQuery asynchronously. Like BaseX it returned an identifier for the executing query and did not accept a callback function or provide a promise. Unlike BaseX however, there were no additional functions to control the asynchronously executing query or obtain its result, rather the asynchronously executing query would run to completion and its result would be discarded, although it may have updated the database via side effects. This facility proved not to be particularly practical and has since been removed. Similarly to BaseX, eXist-db provides a Scheduler XQuery extension module [10] for scheduling the future (or immediate) execution of jobs written in XQuery. Unfortunately even if an XQuery is scheduled for immediate execution, there is no mechanism for obtaining the result of its execution from the initiating XQuery.

eXist-db makes no attempts to avoid side effects during processing, and instead offers many extension functions and a syntax for updating nodes that cause side effects by immediately modifying external state and making the modifications visible. eXist-db also relaxes the XPath deterministic constraint upon Available Documents, and Available Collections, allowing a query to both modify which documents and collections are available (a side effect), and to see changes made by concurrently executing queries.

eXist-db is able to suffer side effects, through making several compromises:

- eXist-db offers the lowest transaction isolation level when executing XQuery - Read Uncommitted.

  eXist-db makes XQuery users somewhat aware of this, and provides XPath extension functions which enable them to lock documents and collections on demand if they require a stronger isolation level.

- eXist-db executes XQuery sequentially as though it was a procedural program.

Whilst some query rewriting is employed to improve performance, eXist-db cannot exploit many of the more advanced optimisations available to functional language compilers: any reordering of the XQuery program's execution path could cause the program to return incorrect results, due to side effects being applied in an order that the XQuery developer had not intended.

Likewise, eXist-db cannot easily parallelise the execution of disjoint statements within an XQuery: as shared-state modified by side effects could introduce race conditions in the XQuery developer's application.

2.3. MarkLogic

MarkLogic provides an XPath extension function named `xdmp:spawn`, which allows another XQuery to be started asynchronously from the calling query. This is done by placing it on the task queue of the MarkLogic task server, and this query may be executed concurrently if the task server has the available resources. The function is *non-blocking*, and for our interests has two modes of operation controlled by an option called `result`. When the *result* option is set to *false*, the calling query has no reference to the queued query, and like eXist-db it can neither retrieve its result, enquire about its status, or abort its execution. When the *result* option is set to *true*, the `xdmp:spawn` function returns what MarkLogic describes as a “value future for the result of the spawned task”. This “value future” is quite unusual, and certainly a vendor extension with no corresponding type in XDM. Essentially, after calling `xdmp:spawn` with the *return* option set to *true*, the calling query continues executing until it tries to access the value of the variable bound to the result of the `xdmp:spawn`, at which point if the *spawned* query has completed executing, the result is available, however if it has not completed then the main query thread blocks and waits for the *spawned* query to complete and provide the result [30]. Similarly to BaseX and eXist-db, MarkLogic also provides mechanisms for the scheduling of XQuery execution through its offline batch processing framework called CPF (Content Processing Framework) [31], and a set of XPath extension functions such as `admin:group-add-scheduled-task` [32].

MarkLogic's *value future* is intriguing in its nature, albeit proprietary. The concept of Futures appear in several programming languages, but unlike other languages (e.g., Java or C++11), MarkLogic's implementation provides no explicit call to *get* the value of the future (possibly with a timeout), instead the *wait* and/or *get* happen as one implicitly when accessing the value through its variable binding.

MarkLogic clearly documents where it allows side effects from within XQuery. There are two distinct types of side effects within MarkLogic, state changes that happen within the scope of the XQuery itself, and those state-changes which are external to the XQuery. For use within the scope of an XQuery, MarkLogic provides an XPath extension function `xdmp:set`, which explicitly

states that it uses “changes to the state (side effects)” [33] to modify the value of a previously declared variable, thus violating the formal semantics of XPath [4]. For modifying state external to an XQuery, MarkLogic provides a series of XPath extension functions for updating nodes and managing documents within the database. Similarly to BaseX, these extension functions do not cause side effects by immediate application, and are invisible to both the executing query and concurrently executing queries [34]. Unlike BaseX, MarkLogic does not implement the XQuery Update specification, but similarly it utilizes a PUL, likewise leading to a process whereby the updates are applied to the database after the query completes and the transaction is committed, thus making the updates visible to subsequent transactions.

Whilst MarkLogic utilizes both a well defined transaction isolation model and deferred updates to mostly avoid side effects within an executing XQuery, we suspect that the use of `xdmp:set` likely places some limitations on possible query optimisations that could be performed.

We have focused on MarkLogic's XQuery implementation, but it is worth noting that MarkLogic also implements XSLT 2.0. All of MarkLogic's XPath extension functions (e.g., `xdmp:set` and `xdmp:insert-*`) are also available from its XSLT processor, and are subject to the same transactional mechanisms as the XQuery processor; therefore our findings are equally applicable to running either XQuery or XSLT on MarkLogic.

2.4. Saxon

Saxon-EE utilises parallel processing in certain specific instances [20]. By default the parsing of input files for the `fn:collection` function is multithreaded, as is the processing of `xsl:result-document` instructions. Note that the outputs produced by multiple `xsl:result-document` instructions are quite independent and never need to be merged; so while this does allow parallel execution of user code and requires careful implementation of features such as try/catch and lazy evaluation, the fact that there is a "fork" with no "join" simplifies things a lot. Furthermore, multi-threading of `xsl:for-each` instructions using a MapReduce approach can be enabled by the user, by setting the `saxon:threads` extension attribute to specify the number of threads to be used.

Saxon-EE allows use of a number of extension functions with side effects, including those in the EXPath File and Binary modules. Similar to the BaseX handling, the Saxon compiler recognises such expressions as causing side effects, and takes a pragmatic approach in attempting to avoid aggressive optimisations which could otherwise disrupt the execution order. Usually instructions in an XSLT sequence constructor will be executed sequentially in the order written, but deviation can be caused by the compiler through lazy evaluation or loop lifting; and this is where problems can arise when side effects are involved. Such optimi-

sations can cause the side effect to happen the wrong number of times (never, or too often), or at the wrong time. It is relatively straightforward to prevent such optimisations for static calls to side-effecting functions, but cannot always be guaranteed for more nested calls, as "side-effecting" is not necessarily recognised as a transitive property. For instance, a function or template which includes a call to a side-effecting function may not itself be recognised as side-effecting. So it is always recommended that side-effecting XPath expressions are "used with care". One mechanism which gives the XSLT author better control when using side-effecting expressions, is the recently added extension instruction `saxon:do`. It is similar to the `xsl:sequence` instruction, but is designed specifically for use when invoking XPath expressions with side effects. In contrast to `xsl:sequence`, when using `saxon:do` any result is always discarded, and the processor ensures that instructions in the sequence constructor are always evaluated sequentially in the order written, avoiding any reordering from optimisations.

As previously mentioned, for use with the Saxon-JS runtime XSLT processor, a number of Interactive XSL extension instructions and functions are available. To enable non-blocking (asynchronous) HTTP requests and document fetching, the `ixsl:schedule-action` instruction is provided. Attributes on the instruction are used to specify an HTTP request, or document URI, and the associated HTTP request is then executed in a new concurrent thread. The callback, for when an HTTP response is returned or the document is fetched (or an HTTP error occurs), is specified using the single permitted `xsl:call-template` child of the `ixsl:schedule-action` instruction. When the `document` attribute has been used, the called template can then access the document(s) using the `fn:doc` or `fn:doc-available` functions; the document(s) will be found in a local cache and will not involve another request to the server. When using the `http-request` attribute, the HTTP response is supplied as the context item to the called template, in the form of an XDM map. Alternatively, `ixsl:schedule-action` can simply be used to start concurrent processing for any action, by using just the `wait` attribute (with a minimal delay). Note that while this provides a "fork", there is no "join", and it is up to the developer to avoid conflicts caused by side effects.

To be able to write interactive applications directly in XSLT, it is necessary to make use of side effects, for example to dynamically update nodes in the HTML page. Almost all of the IXSL extension instructions and functions (such as `ixsl:set-attribute` and `ixsl:set-property` which are used to set attributes on nodes and properties on JavaScript objects respectively) have (or may have) side effects. Note that Saxon-JS runs precompiled XSLT stylesheets, called SEFs (Stylesheet Export Files) generated using Saxon-EE. As described above, during compilation in Saxon-EE, such side-effecting functions and instructions are internally marked as such to prevent optimisations from disrupting the intended execution order.

2.5. xq-promise

Whilst xq-promise [35] is not an implementation of XQuery or XSLT, it is the first known non-vendor specific proposal for a module of XPath extension functions by which XPDL implementations can offer concurrent processing from within an XPDL. It is valuable to review this proposal as theoretically it could be implemented by any XPDL implementation, at present we are only aware of a single implementation for BaseX [36].

xq-promise first and foremost provides a set of XPath extension functions which were inspired by jQuery's Deferred Object utility, it claims to implement the "promise pattern" (see Section 3.5), and focuses on the concept of deferring execution. In its simplest form, the `promise:defer` function takes two parameters: a function of variable arity, and a sequence of arguments of the same arity as the function. Calling `promise:defer` returns a new zero arity function called a "promise", this *promise* function encapsulates the application of the function passed as a parameter to the arguments passed as a parameter. The encapsulation provided by the promise function *defers* the execution of the encapsulated function. The promise function also serves to enable chaining further actions which are dependent on the result of executing the deferred function, such further actions are also deferred. The chaining is implemented through function composition, but is opaque to the user who is provided with the more technically accessible functions `promise:then`, `promise:done`, `promise:always`, `promise:fail`, and `promise:when`.

The functions provided by xq-promise discussed so far allow a user to describe a chain of related actions, where callback functions, for example established through `promise:then`, can be invoked when another function completes with success or failure. Considered in isolation these functions do not explicitly prescribe any asynchronous or concurrent operation. To address this, xq-promise secondly provides an XPath extension function named `promise:fork-join` based on the Fork-join model of concurrency. This functions takes as a parameter a sequence of promise functions, which may then be executed concurrently. The `promise:fork-join` function is a blocking function, which is quite different from those of BaseX, eXist-db, MarkLogic, or Saxon, which are all non-blocking. Rather than scheduling a query for concurrent execution and then returning to the main query so execution can continue, when `promise:fork-join` is invoked *n* query sub-processes are *forked* from the main query which then waits for these to complete, at which point the results of the sub-processes are *joined* together and returned as the result of the function call.

An important insight we offer is that whilst sharing some terminology with implementations in other languages (particularly JavaScript likely due to building upon jQuery's Deferred Object) the *promise* concept used in xq-promise is subtly different [61]. JavaScript Promises upon construction immediately execute the

function that they are provided [38] [39], whereas an xq-promise is not executed until either `promise:fork-join` is used or the promise function is manually applied by the user. Conceptually the xq-promise promises appear to be at odds with the fork-join approach, as once a promise has been constructed, it is likely that useful computation could have been achieved in parallel to the main thread by executing the promise(s) before reaching the fork-join point. The construction of a JavaScript Promise requires an *executor* function, which takes two parameter functions, a resolve function and a reject function. The executor must then call one of these two functions to signal completion. When constructing a promise with xq-promise, completion is instead signalled by the function terminating normally, or raising an XPath error. This may appear to be just syntactical differences, but the distinction is important: the JavaScript approach allows an error value to explicitly be returned upon failure in a functional manner, the xq-promise approach relies instead on `fn:error`... which is a side effect!

On the subject of xq-promise and side effects, xq-promise constructs chains of execution where each step has an dependency on the result of the preceding step. On the surface this may appear similar to how IO Monads (see Section 3.4) compose. The composition of xq-promise through is much more limited, and whilst it ensures some order of execution, its functional semantics are likely not strong enough to ensure a total ordering of execution.

2.6. Conclusion of Implementers Survey

Our conclusion from this survey is twofold. Firstly, all surveyed implementations offer some varying proprietary mechanism for performing asynchronous computations from within a main XPDL thread of execution. A standardised approach is evidently missing from the W3C defined XPDLs, but a requirement has been demonstrated by implementations presumably meeting a technical demand of their users of XPDLs. Secondly, none of the XPDL implementations which we examined adhere strictly to the functional processing semantics required by XPath and/or the respectively implemented XPDL specification. Instead each implementation to a lesser or greater extent offers some operations which cause side effects. Most implementations appear to have taken a pragmatic approach to deliver the features that their users require, often sacrificing the advantages of a pure functional approach to offer a likely more familiar imperative programming model.

3. Solutions offered for non-XPDLs

This survey provides a brief review of several options for non-XPDLs that provide solutions for both concurrent and/or asynchronous execution, and how side effects are managed or avoided. This is not intended as an exhaustive survey, rather the options surveyed herein were subjectively chosen for their variety.

3.1. Actor Model

The Actor Model defines a universal concept, the Actor, which receives messages and undertakes computation in response to a message. Each Actor may also asynchronously send messages to other Actors. A system is typically made up of many of these Actors [40]. Actor systems are another class of embarrassingly parallel problem, as the messages sent between actors are immutable, there is no shared-mutable state to synchronize access to, and so each Actor can run concurrently.

The Actor Model by itself is not enough to clearly describe, manage, or eliminate side-effectful computation, however by nature of its message passing approach it does eliminate the side effects of modifying the shared-state for communication between concurrent threads of execution which is often found in non-actor systems. Through encapsulation, actors may also help to reason about programs with side effects. Systems utilising actors are often built in such a manner that each task specific side-effectful computation is isolated and encapsulated within a single Actor. For example, within an actor system there may only be a single Actor which handles a particular file I/O, then since each Actor likely runs as a separate process, the file I/O has been isolated away from other computation.

The Erlang programming language is possibly the most well known Actor Model like implementation, wherein Actors are known as processes [41]. Erlang itself makes no additional efforts to manage side effects, and additional synchronization primitives are often employed. Within the JVM (Java Virtual Machine) ecosystem, the Akka framework is available for both Java and Scala programming languages [42]. Java as a non-functional language makes no attempts at limiting side effects. Meanwhile, whilst Scala is often discussed as a functional language and does provide many functional programming constructs, it is likely more a general purpose language, as mutability and side effects are not restricted, and it is quite possible to write imperative Scala code. Actor systems are also available for many other programming languages [43], although they do not seem to have gained the same respective popularity as Erlang or Akka.

3.2. Async/Await

The Async/Await concept was first introduced in C#, inspired by F#'s *async workflows* [44], which was in turn inspired by Haskell's Async Monad [45] [46] (see Section 3.4). Async/Await provides syntax extensions to a programming language in the form of the `async` and `await` keywords. Async/Await allows a developer to write a program using a familiar synchronous like syntax but easily achieve asynchronous operation of parts of the program.

Async/Await adds no further processing semantics for concurrency or managing side effects over that of Promises (see Section 3.5), which are often used to implement Async/Await. Async/Await may be thought of as syntactic sugar for

utilising a Promise based implementation, and has recently become very popular with JavaScript developers [47] [48].

3.3. Coroutines

Coroutines are a concept for cooperative multitasking between two (or more) processes within a program. One process within an application, Process A, may *explicitly* yield control to another process, Process B. When control is transferred, the state of Process A is saved, the current state of Process B is restored (or a new state created if there is no previous state), and Process B continues until it *explicitly* yields control back to Process A or elsewhere [49].

Like Actors, the impact of side effects of impure functions can be somewhat isolated within a system by encapsulating them in distinct coroutines. Otherwise Coroutines provide no additional facilities for directly managing side effects, and global state is often shared between them. Unlike Actors, Coroutines are often executed concurrently by means of explicitly yielding control. Without additional control structures, coroutines typically operate on a single-thread, one exception is Kotlin's Coroutines which can be structured to execute concurrently across threads [52].

Some implementations of Coroutines, such as those present in Unity [50], or JavaScript [51], attempt to bring a familiar synchronous programming style to the developer. These implementations typically have a coroutine *yield* multiple results to the caller, as opposed to yielding control. This masks the cooperative multitasking aspect from the developer and presents the return value of a coroutine as an iterable collection of results.

3.4. IO Monads

Haskell is a statically typed, non-strict, pure functional programming language. The *pure* aspect means that every function in Haskell must be *pure*, that is to say akin to a mathematical function in the sense that mathematical functions cannot produce side effects. Even though Haskell prohibits side effects by design, it still enables developers to perform I/O and compute concurrently. This seemingly impassable juxtaposition of academic purism and real-world engineering need is made possible by its IO Monad [54]. Haskell trialled several other approaches in the past, including streams and continuations, before the IO Monad won out as it facilitated a more natural imperative programming style [55].

In Haskell, any function that performs I/O must return an IO type which is monadic. This IO type represents an IO action which has not yet happened. For example if you have a function that reads a string from a file, that function does not directly return a `String`, instead it returns an `IO String`. This is not the result of reading a line from the file, instead it can be thought of as an action that when *executed* will read a line from the file and return a `String`. These IO actions

describe the I/O that you wish to perform, but critically defer its execution. The `IO` actions adhere to *monad laws* which allow them to be composed together. For example given two IO actions, one that reads a file and one that writes a file, they could be composed together into a single IO action which first reads a file and then writes a file, e.g. a copy file IO action.

Importantly, the formal definition for an `IO` type is effectively `IO a = World -> (a, World)`. That is to say that an IO is a state transformation function that takes as input the current state of the world, and produces as the result both a value and a new state of the new world. The `World` is a purely Abstract Data Type, that the Haskell programmer cannot create. The important thing to note here is that the `World` is threaded through the IO function. When multiple IO actions are composed together using monadic application, such as *bind*, the `World` output from a preceding function will be fed to the input of the succeeding function. In this manner the `World` will be threaded through the entire chain of IO actions.

A Haskell program begins by executing a function named `main` that must return an IO, it is typed as `mainIO :: IO ()`. Haskell knows how to execute the IO type function that the main function returns. Naively one can think of this as Haskell's runtime creating the `World` and then calling our IO with it as an argument to execute our code; in reality the Haskell compiler optimises out the `World` during compilation whilst still ensuring the correct execution order. (We may remark that an IO action is similar to a PUL's Update Primitive, and the fact that `main` returns an IO is not dissimilar to an XQuery Update returning both XDM and a PUL.)

By using IO Monads which defer rather than perform I/O, all Haskell functions are pure, and so a Haskell program at evaluation time exhibits no side effects whatsoever, instead finally evaluating to an `IO ()`, i.e. a state transformation function upon the world. As the developer has used monadic composition of their IO actions, this has implicitly threaded the `World` between them, in the order the developer would expect (i.e. in the order of the composition), therefore the state transformation also ensures that the functions are executed in the expected/correct order. At execution time, the machine code representation of the Haskell program is run by a CPU which is side-effecting in nature, and the IO action's side effects are unleashed.

> *It is possible to encapsulate stateful computations so that they appear to the rest of the program as pure (stateless) functions which are guaranteed by the type system to have no interactions whatever with other computations, whether stateful or otherwise (except via the values of arguments and results, of course).*
>
> *—from "State in Haskell", by John Launchbury and Simon Peyton Jones*

Haskell provides further functions for concurrency, but critically these also return IO actions. One such example is `forkIO` with the signature `forkIO :: IO () ->`

`IO ThreadId` [56]. The purpose of forkIO is to execute an IO in another thread, so it takes an IO as an argument, and returns an IO. The important thing to remember here, is that calling the forkIO function does not create a new thread and execute an IO, rather it returns an IO action which describes and defers such behaviour. Later when this IO action is finally executed at run-time, the thread will be created at the appropriate point within the running program. There are also a number of other higher-level abstractions for concurrency in Haskell, such as Async [46], and whilst such abstractions *may* introduce additional monads, they ultimately all operate with IO to defer any non-pure computation. One final point on the IO Monad, is to mention that concurrently executing I/O actions, may at runtime produce side effects that conflict with each other. The IO Monad is only strong enough to ensure correct operation within a single thread of execution, its protections do not cross thread-boundaries. To guard against problems with concurrent modifications additional synchronisation is required. Haskell provides additional libraries of such functions and types for working with synchronization primitives, many of which themselves produce IO actions!

Monads are by no means limited to Haskell, and can likely be used in any language which supports higher-order functions. The preoccupation with Haskell is centred around how it uses Monads to ensure a pure language in the face needing to perform I/O. Several libraries exist which attempt to bring the IO Monad concept to other programming languages, this seems to have been most visible within the Scala ecosystem, where there are now at least five differing established libraries [57]. Whilst all of these efforts are admirable and bring new mechanisms for managing side effects, they all have one weakness which Haskell does not: in Haskell one is forced to ensure that the entire program is pure, because the main function must return an IO. The runtimes of other languages are not structured in this way, and so these IO Monad libraries are forced to rely on workarounds to evaluate the IO. These rely on the user structuring their program around the concept of an IO, and only evaluating that IO as the last operation in their program. For example Monix Task [58], where the user must eventually call `runUnsafeSync` to evaluate the IO, describes the situation as thus:

> *In general prefer to ... structure your logic around asynchronous actions in a non-blocking way. But in case you're blocking only once, in* `main`*, at the "edge of the world" so to speak, then it's OK.*
>
> —*Alexandru Nedelcu*

3.5. Promises and Futures

There may be some confusion over the differences between the computer science terms *Promise*, *Future*, or even *Eventuals*. However, these terms are academically synonymous, as perhaps best explained by Baker and Hewitt, the fathers of the term *Future* [16]:

the mechanism of futures, which are roughly Algol-60 "thunks" which have their own evaluator process ("thinks"?). (Friedman and Wise [18] call futures "promises", while Hibbard [17] calls them "eventuals".)

—Henry G. Baker Jr. and Carl Hewitt

The confusion likely comes from implementations that offer both Future and Promise abstractions to developers looking for safer concurrency facilities, yet use differing terminology and provide vastly different APIs. Two examples of extreme variation of terminology, are the Scala and Clojure programming languages, which each define Future and Promise as distinct classes. The Scala/ Clojure Future class is much more like the computer science definition of Future/ Promise which models computation; whereas the Scala/Clojure Promise class serves a very different purpose, primarily as a memorized data provider for completing a Future class. We are strictly interested in the computer science definition of Promise and Future, and herein will refer to them singly as Promise.

A Promise represents a value which may not yet have been computed. Typically when creating a Promise a computation is immediately started asynchronously and returns a Promise. In implementation terms, a Promise is a reference which will likely take the form of an object, function, or integer. At some point in the future when the asynchronous computation completes, the Promise is fulfilled with the result of the computation which may be either a value or an error. Promises provide developers with an abstraction for concurrent programming, but whether that is executed via cooperative or preemptive multi-tasking is defined by the implementation. Promises by themselves provide no mechanism for avoiding side effects as they are likely eagerly evaluated, with multiple promises being unordered with respect to execution.

Some implementations, for example those based on Promise/A+ like JavaScript, allow you to functionally compose Promises together [53]. This functional composition can allow you to chain together side-effecting functions which are encapsulated within Promises, thus giving an explicit execution order, in a manner not dissimilar to Haskell's IO Monad (see Section 3.4). Unlike Haskell's IO Monad however, this doesn't suddenly mean that your application is pure: remember that JavaScript Promises are eagerly evaluated. It does though offer a judicious JavaScript developer some measure to ensure the correct execution order of her impure asynchronous code.

3.6. Reactive Streams

Reactive Streams enable the composition of a stream of computation, where the Publisher, Subscriber, or a Processor in the stream (which act as both Subscriber and Publisher), may operate asynchronously [59]. A key characteristic of Reactive Streams is that of *back-pressure*, a form of flow control which can prevent slower Subscribers from being overwhelmed by faster asynchronous Producers. This

built-in back-pressure facility appears to be unique to Reactive Streams, and would otherwise have to be manually built by a developer atop other concurrency mechanisms.

The Reactive Streams initiative itself just defines a set of interfaces and principles for Reactive Stream implementations, it is up to the implementations to provide mechanisms for controlling concurrent and parallel processing of streaming values. Typically implementations provide mechanisms for parallelising Processors within a stream, or splitting a stream into many asynchronously executing streams which are later resolved back to the main stream.

Reactive Streams offers little explicitly to help with side effects, however if we consider that a data flow within a non-concurrent stream is always downwards, then streams do provide an almost Monadic-like mechanism for composing processing steps where the order of execution becomes explicit. Likewise, if one was to ensure that the data that is passed from one step to another is immutable, then when there are concurrent or asynchronous Subscribers, there can be no data-driver side effects between them as the data provided by the publisher was immutable, meaning that any changes to the data by a subscriber are isolated to a localised copy of the data.

Examples of Reactive Streams implementations that support concurrent and parallel processing at this time include: RxJava, Akka Streams, Monix, Most.js, and Reactive Streams .NET#

3.7. Conclusion of non-XPDL Solutions Survey

Our survey shows several different options for concurrent/parallel programming. It is possible to build the same application using any of these options, but each offers a different approach and syntax for isolating and managing concurrently executing processes. As well as the underlying computer science principles of each option, the libraries or languages that implement these options can vary between Cooperative Multitasking and Preemptive Multitasking. Coroutines, Async/Await, and Promises are particularly well suited to Cooperative Multitasking systems due to their explicit demarcation of computation boundaries, which can be used to yield the CPU to another process. Likely this is why these options have been adopted in the JavaScript community, where JavaScript Virtual Machines are frequently designed as cooperatively multitasking systems utilising an event loop [60].

We find that the IO Monad is the only surveyed option that is specifically designed to manage computational side effects in a functional manner. This is likely due to the fact that the IO Monad approach was explicitly developed for use in a non-strict purely functional language, i.e. Haskell, whereas all of the other approaches are more generalised, and whilst not explicitly limited to imperative languages are often found in that domain.

Of all the approaches surveyed, to the best of our knowledge, only the development of a Promise-like approach has been realised for XPDLs, namely xq-promise (see Section 2.5). It seems likely that at least aspects of the IO Monad approach (such as that demonstrated by Monix), or Reactive Streams options, could be implemented by utilising XPath extension functions and a written specification of concurrent implementation behaviour, without resorting to proprietary XPath syntax extensions. Conversely, whilst an XPath function based implementation could likely be devised, both Async/Await and Coroutines would likely benefit by extending the XPath language with additional syntax.

In conclusion, we believe that an IO Monad exhibits many of the desirable properties that we set out to discover in Section 1.4. It has strong pure functional properties, strict isolation of side effects, and acts as a building block for constructing further concurrent/parallel processing. Therefore we have chosen to use this as the basis for a solution to handle side effects and sequential or concurrent processing in XPDLs.

4. EXPath Tasks

Herein we describe EXPath Tasks, a module of extension XPath functions for performing *Tasks*. These functions have been designed to allow an XPDL developer to work with both side effects and concurrency in a manner which appears imperative but is functionally pure, and therefore does not require processors to sacrifice optimisation opportunities.

The specification of the functions and their behaviour is defined in Appendix A. We have also developed four reference implementations:

XQuery
: task.xq is written in pure XQuery 3.1 with no extensions. It implements all functions, however all potentially asynchronous operations are executed sychronously. The source code is available from https://github.com/adamretter/task.xq.

XSLT
: task.xsl is written in pure XSLT 3.0 with no extensions. There is a lot of code overlap with task.xq, since much is actually XPath 3.1. Like task.xq, it implements all functions, however all potentially asynchronous operations are executed sychronously. The source code is available from https://github.com/saxonica/expath-task-xslt.

Java
: An implementation of EXPath Tasks for XQuery in eXist-db. The source code is available from https://github.com/eXist-db/exist/tree/expath-task-module-4.x.x/extensions/expath/src/org/expath/task.

JavaScript
: An implementation of EXPath Tasks for XSLT in Saxon-JS.

4.1. The Design of EXPath Tasks

From the findings of our survey on non-XPDL solutions (see Section 3), we felt that the best fit for our requirements (see Section 1.4) was that of developing a module of XPath Functions that could both ensure the correct execution ordering of side-effecting functions, and provide facilities for asynchronous programming.

We decided to adopt the principles of the IO Monad, as we have previously identified it as providing the most comprehensive approach to managing non-deterministic functions in a pure functional language. Our design was heavily influenced by both Haskell's IO [54] and Async [46] packages, and to a lesser extent by Monix's Task [58].

Our decision to develop a module of extension functions rather than grammar extensions, was influenced by a previous monadic approach for XQuery, called *XQuery!*, which utilized grammar extensions but failed to gain adoption [63].

An astute reader may raise the question of why we didn't attempt a translation of IO actions to PUL Update Primitives. The issue that we saw is that a PUL is an opaque collection, which cannot be computed over. With XQuery Update there is no mechanism for directly working with the result of a previous Update Primitive. We required a solution that was applicable to general computation, so we focused on a task based approach. Of course there is the concern that we would have also had to adopt much of the XQuery Update specification to make this work in practice. For XPDLs that are not derived from XQuery this may have been prohibitive to adoption. However, we see no reason why further work could not examine the feasibility of *lifting* a Task to an Update Primitive.

4.1.1. Abstract Data Types

Haskell's IO Monad makes use of an ADT (Abstract Data Type) to represent the *World* which it is transforming. The beauty of using an ADT here is that the Haskell programmer cannot themselves instantiate this type[2], which makes it impossible to execute IO directly. Instead the Haskell compiler is responsible for compiling the application in such a manner that the IO will be implicitly executed at runtime.

Recall that the IO type is really a state transformation function, with the signature

```
IO a = World -> (a, World)
```

To create an equivalent function for XPDLs we need some mechanism for modelling the World ADT. Unfortunately, without requiring *Schema Awareness*, the

[2]Haskell does provide an `unsafePerformIO` function which can *conjure* the world up, and execute the IO. However, such behaviour is considered bad practice in the extreme.

XDM type system is sealed. It is not possible to define new types abstract or otherwise within XPDLs.

To remain within the XPDL specifications we must therefore define the World using some non-abstract existing type. Unfortunately, this means that the developer can also instantiate the World and potentially execute the IO. We developed an initial prototype [62] where we modelled the World simply as an XDM Element named `io:realworld`, thus our XPath IO type function was defined such:

```
declare function io:IO($realworld as element(io:realworld)) as item()+
```

Note the item()+ return type: in XPath there is no tuple type so we have to use a less strict definition than we would prefer. This sequence of items will have 1+*n* items, where the head of the sequence is always the new state of the world (i.e. the XDM element named `io:realworld`), and the tail of the sequence is the result of executing the IO.

Implementations written for XPDLs in non-XPDLs could likely enforce stronger semantics by using some proprietary type outside of the XDM to represent the `World` which is un-instantiable from the XPDL.

Like Haskell's GHC (Glasgow Haskell Compiler), whether there really is a `World` that is present in the application at execution time or not is an implementation detail. Certainly it is crucial that the `World` is threaded through the chain of IO actions at evaluation time to ensure ordering, but implementations are free to optimise the world away as long as they preserve ordering.

4.1.2. Typing a Task

Ultimately we adopted the name *Task* instead of *IO* to represent our embracement of more than just I/O.

The first version of our Task Module was developed around the type definition of a `Task` as:

```
declare function task:task($realworld as element(adt:realworld))
                 as item()+
```

We quickly realised that using this module led to verbose syntax, and that the function syntax obscured the ordering of chains; the ordering of task execution being the most deeply nested and then extending outwards:

```
task:fmap(
    task:fmap(
```

```
        task:value("hello"),
        upper-case#1
    ),
    concat(?, " adam")
)
```

Figure 1. Example of Tasks using Function based syntax

To provide a more natural imperative syntax, we realised that instead of modelling a Task as a function type, we could model it as an XDM Map of functions which can be *applied*. An XDM Map is itself a function from its key to its value. By modelling a Task as Map, we could use the encapsulation concept from OOP (Object Oriented Programming) to place functions in the Task (Map), that act upon that task. Each function that we previously defined that operated upon a Task, we recreated as a function inside the Map which operates on the Task represented by the Map. Thus yielding a fluent imperative-like API that utilises the Map Lookup Operator to appear more familiar to imperative programmers:

```
task:value("hello")
    ? fmap(upper-case#1)
    ? fmap(concat(?, " adam"))
    ? RUN-UNSAFE()
```

Figure 2. Example of Tasks using fluent imperative-like syntax

So our Task type is finalised as:

```
map(xs:string, function(*))
```

More specifically our Task Map is defined as:

```
map {
    'apply': as function(element(adt:realworld)) as item()+,
    'bind': as function($binder as function(item()*) as map(xs:string,
function(*))) as map(xs:string, function(*)),
    'then': as function($next as map(xs:string, function(*))) as
map(xs:string, function(*)),
    'fmap': as function($mapper as function(item()*) as item()*) as
map(xs:string, function(*)),
    'sequence': as function($tasks as map(xs:string, function(*))+) as
map(xs:string, function(*)),
    'async': as function() as map(xs:string, function(*)),
```

```
    'catch': as function($catch as function(xs:QName?, xs:string,
map(*)) as map(xs:string, function(*))) as map(xs:string, function(*)),
    'catches': as function($codes as xs:QName*, $handler as
function(xs:QName?, xs:string, map(xs:QName, item()*)?) as item()*) as
map(xs:string, function(*)),
    'catches-recover': as function($codes as xs:QName*, $handler as
function() as item()*) as map(xs:string, function(*)),
    'RUN-UNSAFE': as function() as item()*
}
```

Observe that the `apply` entry inside the Task map retains our original Task type. The Map provides us with encapsulation which allows for the creation of an imperative-like API. By refactoring our existing Task functions we have been able to preserve both the function syntax-like API and the fluent imperative-like API. This provides developers the opportunity to choose whichever best suits their needs, or to work with a mix of syntaxes as appropriate to them.

4.1.3. Asynchronous Tasks

We provide a mechanism which explicitly allows the developer to state that a Task could benefit from being executed asynchronously. The `task:async` function allows the developer to state their intention, however EXPath Tasks does not specify whether, how, or if this actually executes asynchronously. This gives processors the ability to make informed decisions about concurrent execution based on input from the developer, but great freedom in how that is actually executed. The only constraint on implemetations is that the order of execution within a task chain must be preserved. Developers should rather think of `task:async` as providing a hint to the processor that asynchronous execution would be beneficial, rather than assuming asynchronous execution will always take place.

Conversely, as the only constraint that we place on implementers is that the order of execution within a task chain must be preserved, compliant processors are free to implicitly parallelise operations at execution time providing that constraint holds.

4.1.4. Executing a Task

Recall that a Haskell application starts with a `main` that must return an IO, thus framing the entire application as an IO action. The result of executing an XPDL is always an instance of the XDM (and possibly a PUL). Whilst we could certainly return a Task (map) as the result of the evaluation of our XPDL, what should the processor do when it encounters it? If the processor decides to serialize the XDM then we are likely at the mercy of the W3C XSLT and XQuery Serialization specification, which certainly won't execute our Task by *applying* it to transform the state of the world.

Three potential solutions that present themselves from our research are:

- Prescribe in the specification of EXPath Tasks that an implementation must execute a Task which is returned as the result of the XPDL in a certain manner.
- Incorporate the concept of a PUL into the specification of EXPath Tasks. Each Task would create an Update Primitive which is added into the PUL. The result of evaluating the XPDL would then be both an XDM and a PUL.
- Provide an explicitly unsafe function for evaluating a Task, similar to Haskell's `unsafePerformIO` or Monix Tasks's `runUnsafeSync`.

We decided to adopt a hybrid approach. We provide a `task:RUN-UNSAFE` function, where we explicitly prescribe that this should only appear *once* within an XPDL program, and that it *must* occur at the edge of the program, i.e. as the main function. However, we also explicitly state that implementers are free to override this function. For example, implementations that already support an XQuery Update PUL, may choose to promote a Task chain to a set of Update Primitives when this function is evaluated.

4.2. Using EXPath Tasks

We provide several examples to demonstrate key features of EXPath Tasks.

4.2.1. Composing Tasks

We can use monadic composition to safely compose together several tasks that may at execution time cause side effects, but at evaluation time result in an ordered chain of tasks.

Example 1. Safely Uppercasing a file

```
task:value("/tmp/my-file")
    ?fmap(file:read-text#1)
    ?fmap(fn:upper-case#1)
    ?fmap(fn:write-text("/tmp/my-file-upper", ?))
```

Consider the code in Example 1. We use the EXPath File Module to read the text of a file, we then upper-case the text, and finally write the text out to a new file. We start with a pure value Task holding the path of the source file, by *mapping* this through the `read-text` function a second new task is created. At evaluation time nothing has been executed, instead we have a task that describes that first there is a file path, and then secondly we should read a file from that path. We have composed two operations into one operation which preserves the ordering of the original operations. We then continue by *mapping* through the `upper-`

case, which composes another new task representing all three operations (file path, read-text, and upper-case) in order. Our last mapping composition results in a final new task which represents all four operations in order. When this final task is executed at runtime, each of the four operations will be performed in the correct order.

Through using the EXPath Tasks module, we have safely contained the side effects of the functions from the EXPath File Module, by deferring them from evaluation time to execution time. As the Task is a state transformation, we have also threaded the *World* through our task chain, which ensures that any XPDL processor must execute them in the correct order even in the face of aggressive optimisation.

4.2.2. Using Asynchronous Tasks

We can lift a Task to an Asynchronous Task, which can help provide the XPDL processor with hints about how best to parallelise an XPDL application.

The following is a refactored version of the *fork-join* example from xq-promise [35], to show how concurrent programming can be structured safely using EXPath Tasks.

The example performs 25 HTTP requests to 5 distinct servers and returns the results. First we show the synchronous version:

Example 2. Synchronous HTTP Fetching

```
let $tasks :=
    for $uri in ((1 to 5) !
      ('http://www.google.com', 'http://www.yahoo.com',
       'http://www.amazon.com', 'http://cnn.com',
       'http://www.msnbc.com'))
    let $task :=
        task:value($uri)
            ?fmap(http:send-request(<http:request method="GET" />, ?))
            ?fmap(fn:tail#1)
            ?fmap(fn:trace(?, 'Results found: '))
            ?fmap(function ($res) {
                $res//*:a[@href => matches('^http')]
            })
return
    task:sequence($tasks)
        ?RUN-UNSAFE()
```

Now we show the asynchronous version, where we have only needed to insert two lines of code, the call to `task:async` which lifts each Task into an Asynchronous Task, and a binding to `task:wait-all`:

Example 3. Asynchronous HTTP Fetching

```
let $tasks :=
    for $uri in ((1 to 5) !
      ('http://www.google.com', 'http://www.yahoo.com',
       'http://www.amazon.com', 'http://cnn.com',
       'http://www.msnbc.com'))
    let $task :=
        task:value($uri)
            ?fmap(http:send-request(<http:request method="GET" />, ?))
            ?fmap(fn:tail#1)
            ?fmap(fn:trace(?, 'Results found: '))
            ?fmap(function ($res) {
                $res//*:a[@href => matches('^http')]
            })
            ?async()
return
    task:sequence($tasks)
        ?bind(task:wait-all#1)
        ?RUN-UNSAFE()
```

4.2.3. Using Tasks with IXSL

We now consider how Tasks could be used within an IXSL stylesheet for a Saxon-JS web application. Here we use Tasks to enable both concurrency (an asynchronous HTTP request) and side effects (HTML DOM updates). The code in Example 4 shows an IXSL event handling template for onclick events for the "go" button, and associated functions. The main action of clicking the "go" button is to send an asynchronous HTTP request. The intention is that the HTTP response will provide new content for the `<div id="target">` element in the HTML page, as directed by the local `f:handle-http-response` function. But while awaiting the HTTP response, the "target" `div` is first updated to provide a "Request processing..." message, and the "go" button is hidden; as directed by the local `f:onclick-page-updates` function.

Example 4. Asynchronous HTTP using Tasks in IXSL

```
    <xsl:template match="button[@id eq 'go']" mode="ixsl:onclick">
       <xsl:variable name="onclick-page-updates-task"
                     select="task:of(f:onclick-page-updates#0)"/>
       <xsl:variable name="http-post-task"
                     select="task:of(function(){http:post($request-body,
$request-options)})"/>
       <xsl:variable name="async-http-task"
```

```
                  select="$http-post-task ? fmap(f:handle-http-
response#1) ? async()"/>
      <xsl:sequence select="task:RUN-UNSAFE(task:then($onclick-page-
updates-task, $async-http-task))"/>
   </xsl:template>

   <xsl:function name="f:onclick-page-updates">
      <ixsl:set-style name="display" select="'none'"
                   object="ixsl:page()//button[id='go']"/>
      <xsl:result-document href="#target" method="ixsl:replace-content">
         <p>Request processing...</p>
      </xsl:result-document>
   </xsl:function>

   <xsl:function name="f:handle-http-response">
      <xsl:param name="response" as="map(*)"/>
      <xsl:for-each select="$response?body">
         <xsl:result-document href="#target"
                           method="ixsl:replace-content">
            <p>Response from request:</p>
            <xsl:sequence select="."/>
         </xsl:result-document>
      </xsl:for-each>
      <ixsl:set-style name="display" select="'inline'"
                   object="ixsl:page()//button[id='go']"/>
   </xsl:function>
```

Through using the EXPath Tasks module, we have safely contained the side effects of the local functions. Meanwhile, the use of the `task:async` function allows the Saxon-JS processor to use an asynchronous implementation of the EXPath HTTP Client 2.0 `http:post` function. The task chain is created making use of `task:fmap` to pass the HTTP response to the handler function; and `task:then` to compose the initial `$onclick-page-updates-task` with the main `$async-http-task`, ensuring the correct order for their side effects.

5. Conclusion

In this paper we have surveyed the current state-of-the-art mechanisms by which XPDL processors allow side effects and concurrent programming, and the options available to non-XPDLs for managing side effects and providing concurrent or parallel programming. From this research we have then developed and specified EXPath Tasks, a module of XPath extension functions, that allow developers to safely encapsulate side-effecting functions so that at evaluation time they appear as pure functions and enforce the expected order of execution. Finally, we

have developed several reference implementations of EXPath Tasks to demonstrate the feasability of implementing our specification.

Were the necessary functions available for performing node updates, we believe that the IO Monad approach taken by EXPath Tasks could even have benefits over using XQuery Update. Whilst it provides similarly strong deferred semantics like a PUL, a *PUL* is completely opaque, and one cannot compute over it, unlike a Task chain where Tasks may be composed together.

Whilst at a casual glance it may appear that EXPath Tasks have some similarities to xq-promise, we should be careful to point out that they work quite differently in practice. We believe that EXPath Tasks has the following advantages over xq-promise:

- Correct Ordering of Execution.

  Under aggressive functional optimisation, EXPath Tasks will still preserve the correct order of execution even when tasks have no explicit dependency between them. EXPath Tasks can guarantee the order because they transparently thread the *World* through the chain of computation as tasks are composed, which implicitly introduces dependencies between the Tasks.

- Flexible Asynchronous Processing.

  The asynchronous processing model of EXPath Tasks is very generalised, and only makes guarantees about ordering of execution. This enables many forms of concurrent programming to be expressed using EXPath Tasks, whereas xq-promise only offers *fork-join*. In fact xq-promise can easily be reimplemented atop EXPath tasks, including *fork-join*:

  ```
  declare function local:fork-join($tasks as task:Task(~An)+)
          as task:Task(array(~An)) {
      task:sequence($tasks ! task:async#1)
          ?bind(task:wait-all#1)
  };
  ```

  Interestingly, if the xq-promise API were reimplemented atop EXPath Tasks, it would gain stronger guarantees about execution order.

  Likewise our generalised approach, whilst making explicit the intention of parallelism, does not restrict processors from making further implicit parallelisation optimisations.

- Potential Performance

  An xq-promise Promise is a deferred computation that cannot be executed until its `fork-join` function is called. In comparison EXPath Tasks's Asynchronous Tasks can begin execution at runtime as soon as their construct function is executed, thus making better use of computer resources by starting computation earlier than would be possible in xq-promise.

It will certainly be interesting to see how the XML community responds to our EXPath Tasks specification. We are hopeful that developers working with Tasks need not necessarily have any understanding of Monads to be able to fully exploit the benefits of EXPath Tasks.

We are still at an early stage of investigating how well use of the Task module can be incorporated into IXSL stylesheets for Saxon-JS applications. Does the Task module provide a good solution for handling asynchronous processing and side effects in Saxon-JS? This may only be answerable once more examples have been trialled, once the Saxon-JS implementation is more advanced.

Given an existing Saxon-JS application, a move to use the Task module could involve a significant amount of restructuring. To use the Task module properly, all side-effecting expressions should be wrapped in tasks, and care would need to be taken to chain them together appropriately. Side-effecting expressions are likely to be found in numerous different templates, and so bringing the tasks together could be a challenge, and would likely involve considerable redesign. These challenges are not necessarily a problem with the Task module, but given that currently developers can be relatively free with how side effects and asynchronous processes fit into their XSLT programs; the move to any solution which requires explicit strict management of these is going to be a fairly radical change. But this work would not be without benefit: the current lack of management of side effects can easily result in unexpected results if the developer is not careful. The use of Tasks would eliminate this risk.

Further work is also required to work out exactly how to use Tasks to accomplish some specific actions within a Saxon-JS application. For example, providing a mechanism which allows a user to abort an asynchronous HTTP request. Combining the use of Tasks with IXSL event handling templates, does not seem to work. Instead it seems a solution requires another way to create event listeners from within the XSLT; in which case, perhaps new IXSL extensions are needed.

5.1. Future Work

We have identified several areas for possible future research:

- Stronger/Stricter Explicit Typing

  The explicit types we have specified in our Task Module are not as strict as we would like. This is in general due to a lack of a stronger type system which would allow us to express both abstract and generic types. At run-time the correct types will be inferred by the processor. It would be interesting to research modifications to the XDM so that we can statically express stricter types. For instance, the Saxon processor provides the tuple type [64] syntax extension as a way of defining a more precise type for maps.

  We recognise there may also be an approach where function generation is used, to generate Task functions with stricter types by type switching on

incoming parameters. Due to the large number of types in the XDM to switch over, such generation would itself likely need to be computed.

- Side effects between Concurrent Tasks

  We have provided no mechanisms for avoiding side effects across shared state between parallel tasks at execution time, e.g. race conditions, data corruption, etc. Often such issues can be avoided by developers decomposing asynchronous tasks into smaller asynchronous tasks which have to synchronize via `task:wait-all`, and then begin asynchronously again. A set of functional Task based synchronization primitives which could be used to help in parallel situations would be an interesting extension.

- Additional convenience functions

  Whilst we have provided the building blocks necessary for general computation, additional convenience functions could be added. For instance `gather` (similar to `task:sequence` but with relaxed ordering), `withAsync` (which lifts a normal function into an Asynchronous Task), and `parZip` (which asynchronously zips the results of two tasks together).

  We have provided mechanisms for working with XPath errors, however we could also consider functions for working with *error values*. We see no reason why something akin to an Either (disjoint union) could not be developed to work with EXPath Tasks, where a result is *either* an error value or the result of successful computation.

A. EXPath Tasks Module Definitions

A.1. Namespaces and Prefixes

This module makes use of the following namespaces to contain its application. The URIs of the namespaces and the conventional prefixes associated with them are:

- `http://expath.org/ns/task` for functions -- associated with `task`.
- `http://expath.org/ns/task/adt` for abstract data types -- associated with `adt`.

A.2. Types

As an attempt at simplifying the written definition of the functions within the Task Module, we have specified a number of type aliases. The concrete types are likely of little interest to users of the Task Module who are more concerned with behaviour than implementation detail. Implementers which need such detail may substitute the aliases for the concrete types as defined below.

We have followed the XPath convention of using lower-cased names for our functions, apart from `task:RUN-UNSAFE` where the use of continuous capital letters is intended to draw developer attention. Our type aliases are described using a capitalised-cased naming convention to visually distinguish them from function names.

Alias	**Concrete Type**
~A	The `~` signifies that this is a generic type, and the `A` is just a placeholder for the actual type. Concretely this is at least an `item()*`, however intelligent processors can likely infer and enforce stricter types through the functionally composed Task chain.

<table>
<tr><th>Alias</th><th>Concrete Type</th></tr>
<tr><td>

task:Task(~A)

</td><td>

The `task:Task` type alias, is concretely `map(xs:string, function(*))`.
The inner aliased generic type, indicates that the Task when executed returns a result of type `~A`.
Specifically the Task map has the following non-optional entries:

```
map {
    'apply': as function(World) as item()+,
    'bind': as function($binder $binder as function(~A) as
task:Task(~B)) as task:Task(~B),
    'then': as function($next as task:Task(~B)) as
task:Task(~B),
    'fmap': as function($mapper as function(~A) as ~B) as
task:Task(~B),
    'sequence': as function($tasks as task:Task(~An)+) as
task:Task(array(~An)),
    'async': as function() as task:Task(task:Async(~A)),
    'catch': as function($catch as function(xs:QName?,
xs:string, map(*)) as task:Task(~B)) as task:Task(~B),
    'catches': as function($codes as xs:QName*, $handler as
function(xs:QName?, xs:string, map(xs:QName, item()*)?) as
~B) as task:Task(~B),
    'catches-recover': as function($codes as xs:QName*,
$handler as function() as ~B) as task:Task(~B),
    'RUN-UNSAFE': as function() as ~A
}
```

Note: Each of the functions defined in the Task Map have the exact same behaviour as their cousins of the same name residing outside of the map. The only difference is that the functions inside the Map don't need an explicit task argument.

</td></tr>
</table>

<table>
<tr><th>Alias</th><th>Concrete Type</th></tr>
<tr><td>task:ErrorObject</td><td>The <code>task:ErrorObject</code> type alias, is concretely <code>map(xs:QName, item()*)</code>.
All entries in the map are optional, but otherwise it is structured as:
<pre>map {
    xs:QName("err:value") : item()*,
    xs:QName("err:module") : xs:string?,
    xs:QName("err:line-number") : xs:integer?,
    xs:QName("err:column-number") : xs:integer?
    xs:QName("err:additional") : item()*
}</pre></td></tr>
<tr><td>task:Async(~A)</td><td>The <code>task:Async</code> type alias, is concretely <code>function(element(adt:scheduler)) as ~A</code>.
The inner aliased generic type, indicates that the Async if it runs to completion will compute a result of type <code>~A</code>.</td></tr>
</table>

A.3. Functions

A.3.1. Basic Task Construction

This group of functions offer facilities for constructing basic tasks. They usually form the starting point of a task chain.

A.3.1.1. task:value

Summary: Constructs a Task from a *pure* value.

Signature: `task:value($v as ~A) as task:Task(~A)`.

Rules: When the task is run it will return the value of *$v*.

Notes: In Haskell this would be known as `return` or sometimes alternatively `unit`.

In Scala Monix this would be known as `now` or `pure`.

In formal descriptive terms this is:

```
value :: a -> Task a
```

Example **Example A.1. Task from a String**

```
task:value("hello world")
```

A.3.1.2. task:of

Summary Constructs a Task from a function.

This provides a way to wrap a potentially non-pure (i.e. side-effecting) function and delay its execution until the Task is executed.

Signature `task:of($f as function() as ~A) as task:Task(~A).`

Rules The function is lifted into the task, which is to say that the function will not be executed until the task is executed. When the task is run, it will execute the function and return its result.

Notes In Haskell there is no direct equivalent.

In Scala Monix this would be known as `eval` or `delay`.

In formal descriptive terms this is:

```
of :: (() -> a) -> Task a
```

Example **Example A.2. Task which computes the system time from a side-effecting function.**

```
task:of(util:system-time#0)
```

A.3.2. Task Composition

This group of functions offer facilities for functionally composing tasks together.

A.3.2.1. task:bind

Summary Composes a new Task from an existing task and a binder function which creates a new task from the existing task's value.

Signature `task:bind($task as task:Task(~A), $binder as function(~A) as task:Task(~B)) as task:Task(~B).`

Rules When the resultant task is executed, the binder function processes the existing task's value, and then the result of the task is returned.

Notes In Haskell this is also called `bind` and often written as >>=.

In Scala Monix this is known as `flatMap`.

In formal descriptive terms this is:

```
bind :: Task a -> (a -> Task b) -> Task b
```

Examples

Example A.3. Using bind to Square a number

```
task:bind(task:value(99), function($v) {
    task:value($v * $v)
})
```

Example A.4. Using bind to Transform a value

```
task:bind(task:value("hello"), function($v) {
    task:value(fn:upper-case($v))
})
```

Example A.5. Using bind to conditionally raise an error

```
task:bind(task:value("hello"), function($v) {
    if ($v eq "goodbye")
    then
        task:error((), "It's not yet time to say goodbye!",
())
    else
        task:value($v)
})
```

Example A.6. Using bind to compose two tasks

```
let $task1 := task:value("hello")
let $task2 := task:value("world")
return
    task:bind($task1, function($v1) {
        task:bind($task2, function($v2) {
            task:value($v1 || " " || $v2)
        })
    })
```

A.3.2.2. task:then

Summary: Composes a new Task from an existing task and a new task. It is similar to `task:bind` but discards the existing task's value.

Signature: `task:then($task as task:Task(~A), $next as task:Task(~B)) as task:Task(~B).`

Rules: When the resultant task is executed, the existing task is executed and the result discarded, and then the result of the next task is returned.

`task:then($task, $next)` is equivalent to `task:bind($task, function($_) { $next }).`

Notes: In Haskell this is also a form of bind which is sometimes called `then`, and often written as `>>`.

In Scala Monix this is direct equivalent.

In formal descriptive terms this is:

```
then :: Task a -> (_ -> Task b) -> Task b
```

Example: **Example A.7. Sequentially composing two tasks**

```
task:then(task:value("something we don't further need"),
task:value("something important"))
```

A.3.2.3. task:fmap

Summary: Composes a new Task from an existing task and a mapping function which creates a new value from the existing task's value.

Signature: `task:fmap($task as task:Task(~A), $mapper as function(~A) as ~B) as task:Task(~B).`

Rules: When the resultant task is executed, the mapper function processes the existing task's value, and then the result of the task is returned.

Notes: In Haskell this is also called `fmap` and often written as `<$>`.

In Scala Monix this is known as `map`.

In formal descriptive terms this is:

```
fmap :: Task a -> (a -> b) -> Task b
```

Examples

Example A.8. Upper-casing a Task String

```
task:fmap(task:value("hello"), fn:upper-case#1)
```

Example A.9. Concatenating a Task String

```
task:fmap(task:value("hello"), fn:concat(?, " world"))
```

Example A.10. Extracting the code-points of a Task String (e.g. type conversion, String to Integer+)

```
task:fmap(task:value("hello"), fn:string-to-codepoints#1)
```

A.3.2.4. task:sequence

Summary: Constructs a new Task representating the sequential application of one or more other tasks.

Signature: `task:sequence($tasks as task:Task(~An)+) as task:Task(array(~An))`.

Rules: When the resultant task is executed, each of the provided tasks will be executed sequentially, and the results returned as an XDM array. The order of entries in the resultant array is the same as the order of *$tasks*.

Notes: In Haskell and Scala Monix this is known as `sequence`.

In formal descriptive terms this is:

```
sequence :: [Task a] -> Task [a]
```

Examples

Example A.11. Sequencing three Tasks into one

```
task:sequence((task:value("hello"), task:value(54),
task:value("goodbye"))
```

A.3.3. Task Error Management

This group of functions offers facilities for using tasks in the face of XPath errors. Several can be used along with `task:error` as a form of conditional branching or downward flow control.

A.3.3.1. task:error

Summary
: Constructs a Task that raises an error.

This is a Task abstraction for `fn:error`.

Signature
: `task:error($code as xs:QName?, $description as xs:string, $error-object as task:ErrorObject?) as task:Task(none).`

Rules
: The error is not raised until the task is run.

The parameters *$code*, and *$description* have the same purpose as those with the same name defined for `fn:error`.

The parameter *$error-object* has the same purpose but is a type restriction of the parameter with the same name defined for `fn:error`, it should be of type task:ErrorObject.

Notes
: In Haskell this would be closest to `fail`.

In Scala Monix this would be known as `raiseError`.

In formal descriptive terms this is:

```
error :: (code, description, error-object) -> Task none
```

Examples

Example A.12. Constructing a simple Task Error

```
task:error(xs:QName("local:error001"), "BOOM!", ())
```

A.3.3.2. task:catch

Summary
: Constructs a Task which catches any error raised by another task.

This is similar to `task:catches` except that all errors are caught.

Signature
: `task:catch($task as task:Task(~A), $handler as function(xs:QName?, xs:string, task:ErrorObject?) as task:Task(~B)) as task:Task(~B).`

Rules
: When the resultant task is executed, the handler function catches any error from executing the existing task, and then the result of the handler task is returned.

The handler function accepts three arguments, the first is the QName of the error that was caught, the second is the description of the error that was caught, and the third are the ancillary error details collected as a `task:ErrorObject`.

If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes

In Haskell this is similar to `catch`.

In Scala Monix this would be similar to `onErrorHandleWith`.

In formal descriptive terms this is:

```
catches :: Task a -> ([code, description, errorObject] -> Task b) -
> Task b
```

Example

Example A.13. Using catch to recover from an error

```
let $my-error-code := xs:QName("local:error01")
return
    task:catch(task:error($my-error-code, "Boom!", ()),
function($actual-code, $actual-description, $actual-error-
object) {
        "Handled error: " || $actual-code
    })
```

A.3.3.3. task:catches

Summary

Constructs a Task which catches specific errors of another task.

This is similar to `task:catch-recover` except that the error handler receives details of the error.

Signature

`task:catches($task as task:Task(~A), $codes as xs:QName*, $handler as function(xs:QName?, xs:string, task:ErrorObject?) as ~B) as task:Task(~B).`

Rules

When the resultant task is executed, the handler function catches any matching errors identified by the parameter *$codes* from executing the existing task, and then the result of the handler task is returned.

The handler function accepts three arguments, the first is the QName of the error that was caught, the second is the description of the error that was caught, and the third are the ancillary error details collected as a `task:ErrorObject`.

If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes

In Haskell this is similar to `catches`.

In Scala Monix this would be similar to `onErrorHandle`.

In formal descriptive terms this is:

```
catches :: Task a -> ([code] -> ([code, description, errorObject] ->
```

```
b)) -> Task b
```

Example

Example A.14. Using catches to recover from an error

```
let $my-error-code := xs:QName("local:error01")
return
    task:catches(task:error($my-error-code, "Boom!", ()),
($my-error-code, xs:QName("err:XPDY004")), function($actual-
code, $actual-description, $actual-error-object) {
        "Handled error: " || $actual-code
    })
```

A.3.3.4. task:catches-recover

Summary
: Constructs a Task which catches specific errors of another task.

  This is similar to `task:catches` except that the error handler does not receive details of the error.

Signature
: `task:catches-recover($task as task:Task(~A), $codes as xs:QName*, $handler as function() as ~B) as task:Task(~B).`

Rules
: When the resultant task is executed, the handler function catches any matching errors identified by the parameter *$codes* from executing the existing task, and then the result of the handler task is returned.

  If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes
: In Haskell this is similar to `catches`, but it does not pass the error details to the *$handler*.

  In Scala Monix this would be similar to `onErrorRecover`.

  In formal descriptive terms this is:

```
catches-recover :: Task a -> ([code] -> (\_ -> b)) -> Task b
```

Example

Example A.15. Using catches-recover to recover from an error

```
let $my-error-code := xs:QName("local:error01")
return
    task:catches-recover(task:error($my-error-code,
"Boom!", ()), ($my-error-code), function() {
        "Recovering from error..."
```

```
})
```

A.3.4. Asynchronous Tasks

This group of functions offers facilities for constructing asynchronous tasks and acting upon their progress.

A.3.4.1. task:async

Summary
: Constructs an Asynchronous Task from an existing Task.

Signature
: `task:async($task as task:Task(~A)) as task:Task(task:Async(~A))`.

Rules
: The existing task will be composed into a new task which may be executed asynchronously.

This function makes no guarantees about how, when, or if the asynchronous task is executed other than the fact that execution will not begin before the task itself is executed.

Implementations are free to implement asynchronous tasks using any mechanism they wish including cooperative multitasking, preemptive multitasking, or even plain old single-threaded synchronous. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

When the task is run, it may start an asynchronous process which executes the task, regardless it returns a reference to the (possibly) asynchronous process, which may later be used for cancellation or obtaining the result of the task.

If the function call results in asynchronous behaviour (i.e. a fork of the execution path happens), then the asynchronous task inherits the *Static Context*, and a copy of the *Dynamic Context* where the *Context item*, *Context position*, and *Context size* have been reinitialised. If an implementation supports XQuery Update PUL, then any Update Primitives generated in the Asynchronous Task are merged back to the main Task only when `task:wait` or `task:wait-all` is employed.

Notes
: In Haskell this is similar to `async` from the `Control.Concurrent.Async` package.

In Scala Monix this would be known as `executeAsync`.

In formal descriptive terms this is:

```
async :: Task a -> Task (Async a)
```

Example **Example A.16. Task which asynchronously posts a document**

```
task:async(
    task:fmap(
        task:value("http://somewebsite.com"),
        http:post(?, <some-document/>)
    )
)
```

A.3.4.2. task:wait

Summary Given an Async this function will extract its value and return a Task of the value.

Signature `task:wait($async as task:Async(~A)) as task:Task(~A).`

Rules At execution time of the task returned by this function, if the Asynchronous computation represented by the *$async* reference has not yet completed, then this function will block until the asynchronous computation completes.

This function makes no guarantees about how, when, or if blocking occurs other than the fact that any blocking (if required) will not begin before the task itself is executed.

Implementations are free to implement waiting upon asynchronous tasks using any mechanism they wish. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes In Haskell this is similar to `wait` from the `Control.Concurrent.Async` package.

In Scala Monix this would be similar to `Await.result`.

In formal descriptive terms this is:

```
wait :: Async a -> Task a
```

Example **Example A.17. Task waiting on an asynchronous task**

```
let $async-task :=
```

```
        task:async(
            task:fmap(
                task:value("http://somewebsite.com"),
                http:post(?, <some-document/>)
            )
        )
    return

        (: some further task chain of processing... :)

        (: wait on the asynchronous task to complete :)
        task:bind(
            $async-task,
            task:wait#1
        )
```

A.3.4.3. task:wait-all

Summary: Given multiple Asyncs this function will extract their values and return a Task of the values.

Signature: `task:wait-all($asyncs as array(task:Async(~A))) as task:Task(array(~A))`.

Rules: At execution time of the task returned by this function, if any of the Asynchronous computations represented by the *$asyncs* references have not yet completed, then this function will block until all the asynchronous computations complete.

This function makes no guarantees about how, when, or if blocking occurs other than the fact that any blocking (if required) will not begin before the task itself is executed.

Implementations are free to implement waiting upon asynchronous tasks using any mechanism they wish. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

This is equivalent to:

```
task:bind($task, function($asyncs as array(*)) as
map(xs:string, function(*)) {
    task:sequence(array:flatten(array:for-each($asyncs,
task:wait#1)))
})
```

Notes

In Haskell there is no direct equivalent, but it can be modelled by a combination of `wait` and `sequence`.

In Scala Monix there is no direct equivalent.

In formal descriptive terms this is:

```
wait-all :: [Async a] -> Task [a]
```

Example

Example A.18. Task waiting on multiple asynchronous tasks

```
let $async-tasks :=
    (
        task:async(
            task:fmap(
                task:value("http://websiteone.com"),
                http:post(?, <some-document/>)
            )
        ),
        task:async(
            task:fmap(
                task:value("http://websitetwo.com"),
                http:post(?, <some-document/>)
            )
        )
    )
return

    (: some further task chain of processing... :)

    (: wait for all asynchronous tasks to complete :)
    task:bind(
        task:sequence($async-tasks),
        task:wait-all#1
    )
```

A.3.4.4. task:cancel

Summary	Given an Async this function will attempt to cancel the asynchronous process.
Signature	`task:cancel($async as task:Async(~A)) as task:Task().`
Properties	This function is *non-blocking*.
Rules	At execution time of the task returned by this function, cancellation of the Asynchronous computation represented by the *`$async`* reference may be attempted.

This function makes no guarantees about how, when, or if cancellation occurs other than the fact that any cancellation (if required/possible) will not begin before the task itself is executed. Regardless the Asynchronous reference is invalidated by this function.

Implementations are free to implement cancellation of asynchronous tasks using any mechanism they wish, they are also free to ignore cancellation as long as the Asynchronous reference is still invalidated. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes — In Haskell this is similar to `cancel` from the `Control.Concurrent.Async` package.

In Scala Monix this is known as `cancel`.

In formal descriptive terms this is:

```
cancel :: Async a -> Task ()
```

Example

Example A.19. Cancelling an asynchronous task

```
let $async-task :=
    task:async(
        task:fmap(
            task:value("http://somewebsite.com"),
            http:post(?, <some-document/>)
        )
    )
return

    (: some further task chain of processing... :)

    (: cancel the asynchronous task :)
    task:bind(
        $async-task,
        task:cancel#1
    )
```

A.3.4.5. task:cancel-all

Summary — Given multiple Asyncs this function will attempt to cancel all of the asynchronous processes.

Signature `task:cancel-all($asyncs as array(task:Async(~A))) as task:Task()`.

Properties This function is *non-blocking*.

Rules At execution time of the task returned by this function, cancellation of all Asynchronous computations represented by the *$asyncs* references may be attempted.

This function makes no guarantees about how, when, or if cancellation occurs other than the fact that any cancellation (if required/possible) will not begin before the task itself is executed. Regardless the Asynchronous references are invalidated by this function.

Implementations are free to implement cancellation of asynchronous tasks using any mechanism they wish, they are also free to ignore cancellation as long as the Asynchronous references are still invalidated. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes In Haskell there is no direct equivalent, but it can be modelled by a combination of `cancel` and `sequence`.

In Scala Monix there is no direct equivalent.

In formal descriptive terms this is:

```
cancel-all :: [Async a] -> Task ()
```

Example **Example A.20. Cancelling asynchronous tasks**

```
let $async-tasks :=
    (
        task:async(
            task:fmap(
                task:value("http://websiteone.com"),
                http:post(?, <some-document/>)
            )
        ),
        task:async(
            task:fmap(
                task:value("http://websitetwo.com"),
                http:post(?, <some-document/>)
            )
        )
    )
```

```
return

    (: some further task chain of processing... :)

    (: cancel all asynchronous tasks :)
    task:bind(
        task:sequence($async-tasks),
        task:cancel-all#1
    )
```

A.3.5. Unsafe Tasks

This defines a single function `task:RUN-UNSAFE`, which is useful only when a task chain needs to be executed. If an XPDL implementation cannot provide a better mechanism, then this may be implemented and used as a last resort.

A.3.5.1. task:RUN-UNSAFE

Summary: Executes a Task Chain and returns the result.

This function is **inherently unsafe**, as it causes any side effects within the Task chain to be actualised.

If this function is used within an application, it should only be invoked once, and it should be at the edge of the application, i.e. in the position where it is the first and only thing to be directly executed by the application at runtime. No further computation, neither on the result of this function, or after this function call should be attempted by the application.

Signature: `task:RUN-UNSAFE($task as task:Task(~A)) as ~A`.

Properties: This function is *nondeterministic*.

Rules: At execution time, the task chain is evaluated and the result returned.

However, if implementations can provide a safer mechanism for the execution of a Task after the XPDL has completed evaluation, then they are free to override this as they see fit. Once such mechanism could be to promote the Task chain to a set of Update Primitives within a PUL and then demote this to an identity function.

Notes: In Haskell the closest equivalent is `unsafePerformIO`.

In Scala Monix the closest approach would be a combination of `runToFuture` and `Await.result`.

In formal descriptive terms this is:

```
RUN-UNSAFE :: Task a -> a
```

Example

Example A.21. Unsafely executing a Task

```
(:~
 : Just a utility function for calculating
 : previous sightings of Halley's comet
 :)
declare function local:halleys-sightings($before-year) {
    let $start := 1530
    let $interval := 76

    for $range in
        ($start - $interval to $before-year - $interval)
    let $visible := $range + $interval
    where (($visible - $start) mod $interval) eq 0
    return
        $visible
};

let $task := task:fmap(
    task:fmap(
        task:of(util:system-time#0),
        fn:year-from-date#1
    ),
    local:halleys-sightings#1
)
return

    task:RUN-UNSAFE($task)
```

Bibliography

[1] James Clark. Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C Recommendation 16 November 1999 (Status updated October 2016). 1999-11-16. https://www.w3.org/TR/1999/REC-xpath-19991116/.

[2] Anders Berglund. Scott Boag. Mary Fernández. Scott Boag. Michael Kay. Jonathan Robie. Jérôme Siméon. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C Recommendation 14 December 2010 (Link errors corrected 3 January 2011; Status updated October 2016). 2010-12-14. https://www.w3.org/TR/xpath20/.

[3] Mary Fernandez. K Karun. Mark Scardina. *XPath Requirements Version 2.0*. W3C Working Draft 3 June 2005. 2005-06-03. https://www.w3.org/TR/xpath20req/.

[4] Denise Draper. Peter Fankhauser. Mary Fernández. Ashok Malhotra. Kristoffer Rose. Michael Rys. Jérôme Siméon. Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*. W3C Recommendation 14 December 2010 (revised 7 September 2015). 2015-09-07. https://www.w3.org/TR/xquery-semantics/.

[5] Steve Muench. Mark Scardina. *XSLT Requirements Version 2.0*. W3C Working Draft 14 February 2001. 2001-02-14. https://www.w3.org/TR/xslt20req/.

[6] Don Chamberlin. Peter Fankhauser. Massimo Marchiori. Jonathan Robie. *XML Query (XQuery) Requirements*. W3C Working Group Note 23 March 2007. 2007-03-27. https://www.w3.org/TR/xquery-requirements/.

[7] John Snelson. Jim Melton. *XQuery Update Facility 3.0. Pending Update Lists.* W3C Working Group Note 24 January 2017. 2017-01-24. https://www.w3.org/TR/xquery-update-30/#id-pending-update-lists.

[8] Christian Grün. BaseX. 2018-10-31T16:11:00Z. *Jobs Module*. BaseX. http://docs.basex.org/wiki/Jobs_Module.

[9] Adam Retter. eXist-db. *eXist-db Util XQuery Module*. Git Hub. http://www.exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/util&location=java:org.exist.xquery.functions.util.UtilModule&details=true.

[10] Adam Retter. eXist-db. *eXist-db Scheduler XQuery Module*. Git Hub. http://www.exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/scheduler&location=java:org.exist.xquery.modules.scheduler.SchedulerModule.

[11] IBM. 2012-03-07. *IBM100 - Power 4 : The First Multi-Core, 1GHz Processor*. https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

[12] Michael Perrone. 2009. *Multicore Programming Challenges*. IBM, TJ Watson Research Lab. 978-3-642-03868-6. 10.1007/978-3-642-03869-3_1. Springer. https://link.springer.com/chapter/10.1007%2F978-3-642-03869-3_1. *Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science*. 5704.

[13] Pedro Fonseca. Cheng Li. Rodrigo Rodrigues. 2011-04-10. *Finding complex concurrency bugs in large multi-threaded applications*. *EuroSys '11 Proceedings of the sixth conference on Computer systems*. 215-228. ACM. 978-1-4503-0634-8. 10.1145/1966445.1966465. https://dl.acm.org/citation.cfm?id=1966465.

[14] Matthew Loring. Mark Marron. Daan Leijen. 2017-10-24. *Semantics of Asynchronous JavaScript*. *DLS 2017 Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages table of contents*. 51-62. ACM. 978-1-4503-5526-1. 10.1145/3133841.3133846. https://dl.acm.org/citation.cfm?id=3133846.

[15] Cosmin Radoi. Stephan Herhut. Jaswanth Sreeram. Danny Dig. 2015-01-24. *Are Web Applications Ready for Parallelism?*. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 289-290. ACM. 978-1-4503-3205-7. 10.1145/2688500.2700995. https://dl.acm.org/citation.cfm?id=2700995.

[16] Henry G. Baker, Jr.. Carl Hewitt. 1977-08-15. *The Incremental Garbage Collection of Processes*. *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. 55-59. ACM. 10.1145/800228.806932.

[17] Peter Hibbard. 1976. *Parallel Processing Facilities*. *New Directions in Algorithmic Languages*. 1-7.

[18] Daniel Friedman. David Wise. 1976. *The Impact of Applicative Programming on Multiprocessing*. *International Conference on Parallel Processing 1976*. 263–272. ACM.

[19] Jeffrey Dean. Sanjay Ghemawat. 2004. *MapReduce: Simplified Data Processing on Large Clusters*. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. 137-150. https://research.google.com/archive/mapreduce-osdi04.pdf.

[20] Michael Kay. 2015-02-14. *Parallel Processing in the Saxon XSLT Processor*. *XML Prague 2015 Conference Proceedings*. 978-80-260-7667-4. http://www.saxonica.com/papers/xmlprague-2015mhk.pdf.

[21] Jonathan Robie. 2016-03-03T12:05:03-05:00. EXPath Mailing List. *Re: [expath] Re: New Modules? Promise Module, Async Module*. https://groups.google.com/forum/#!msg/expath/Isjeez-5op4/-DCn-KJGBAAJ.

[22] O'Neil Delpratt. Michael Kay. 2013-08-06. *Interactive XSLT in the browser*. *Balisage Series on Markup Technologies, vol. 10 (2013)*. 10. https://doi.org/10.4242/BalisageVol10.Delpratt01. https://www.balisage.net/Proceedings/vol10/html/Delpratt01/BalisageVol10-Delpratt01.html.

[23] Adam Retter. 2018-10-03. *EXPath and Asynchronous HTTP*. https://blog.adamretter.org.uk/expath-and-asynchronous-http/.

[24] Jesús Camacho-Rodríguez. Dario Colazzo. Ioana Manolescu. 2015. *PAXQuery: Efficient Parallel Processing of Complex XQuery*. *IEEE Transactions on Knowledge and Data Engineering*. Institute of Electrical and Electronics

Engineers. 1977-1991. 10.1109/TKDE.2015.2391110. https://hal.archives-ouvertes.fr/hal-01162929/document.

[25] Ghislain Fourny. Donald Kossmann. Markus Pilman. Tim Kraska. Daniela Florescu. Darin Mcbeath. *WWW 2009 MADRID! Track: XML and Web Data / Session: XML Querying XQuery in the Browser*. 2009-04-20. *XQuery in the Browser*. http://www2009.eprints.org/102/1/p1011.pdf.

[26] Philip Fennell. 2013-06-15. *XML London 2013 Conference Proceedings*. 1. 978-0-9926471-0-0. *Extremes of XML*. https://xmllondon.com/2013/xmllondon-2013-proceedings.pdf#page=80.

[27] Jirka Kosek. John Lumley. 2013-12-03. *Binary Module 1.0*. EXPath. http://expath.org/spec/binary/1.0.

[28] Christian Grün. 2015-02-20. *File Module 1.0*. EXPath. http://expath.org/spec/file/1.0.

[29] BaseX. 2018-08-26T16:13:04Z. *Concepts: Pending Update List*. BaseX. http://docs.basex.org/wiki/XQuery_Update#Pending_Update_List.

[30] MarkLogic. 2018. *xdmp:spawn — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/xdmp:spawn?q=spawn&v=9.0&api=true.

[31] MarkLogic. 2018. *Developing Modules to Process Content (Content Processing Framework Guide) — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/guide/cpf/modules.

[32] MarkLogic. 2018. *admin:group-add-scheduled-task — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/admin:group-add-scheduled-task.

[33] MarkLogic. 2018. *xdmp:set — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/xdmp:set.

[34] MarkLogic. 2018. *Understanding Transactions in MarkLogic Server (Application Developer's Guide) — MarkLogic 9 Product Documentation*. *Visibility of Updates*. MarkLogic. https://docs.marklogic.com/guide/app-dev/transactions#id_85012.

[35] James Wright. 2016-02-13. *XML Prague 2016 Conference Proceedings*. 1. 978-80-906259-0-7. *Promises and Parallel XQuery Execution*. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#page=151.

[36] James Wright. *xq-promise*. 2016-04-29. Git Hub. https://github.com/james-jw/xq-promise.

[37] Conway Melvin. 1963. *A Multiprocessor System Design*. ACM. 10.1145/1463822.1463838. *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. 139-146.

[38] 2018-11-15T06:49:39Z. *Promise | MDN*. *Syntax*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Syntax.

[39] 2015. 6th Edition. *Standard ECMA-262*. *ECMAScript® 2015 Language Specification*. Ecma International. http://www.ecma-international.org/ecma-262/6.0/#sec-promise-executor.

[40] Carl Hewitt. Peter Bishop. Richard Steiger. 1973. *A Universal Modular ACTOR Formalism for Artificial Intelligence*. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. 235-245. Morgan Kaufmann Publishers Inc.. http://dl.acm.org/citation.cfm?id=1624775.1624804.

[41] Joe Armstrong. Ericsson AB. 2007. *A History of Erlang*. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. 6-1--6-26. ACM. 978-1-59593-766-7. 10.1145/1238844.1238850.

[42] Lightbend, Inc.. *Akka Documentation*. *Actors*. 2018-12-07T11:55:00Z. https://doc.akka.io/docs/akka/2.5.19/actors.html.

[43] 2019-01-17T21:11:00Z. *Actor model*. *Actor libraries and frameworks*. Wikipedia. https://en.wikipedia.org/wiki/Actor_model#Actor_libraries_and_frameworks.

[44] Anders Hejlsberg. Microsoft. 2010-10-28T10:13:00Z. Channel 9. *Introducing Async – Simplifying Asynchronous Programming*. https://channel9.msdn.com/Blogs/Charles/Anders-Hejlsberg-Introducing-Async.

[45] Don Syme. Microsoft Research. 2007-10-10. *Introducing F# Asynchronous Workflows*. https://blogs.msdn.microsoft.com/dsyme/2007/10/10/introducing-f-asynchronous-workflows/.

[46] Simon Marlow. 2012. *async-2.2.1: Run IO operations asynchronously and wait for their results*. *Control.Concurrent.Async*. Hackage. http://hackage.haskell.org/package/async/docs/Control-Concurrent-Async.html.

[47] Mostafa Gaafar. 2017-03-26. *6 Reasons Why JavaScript's Async/Await Blows Promises Away (Tutorial)*. Hacker Noon. https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9.

[48] Ilya Kantor. 2019. *Promises, async/await*. *Async/await*. JavaScript.info. https://javascript.info/async-await.

[49] Melvin Conway. 1963-07. *Design of a Separable Transition-diagram Compiler*. *ACM Communications*. 6. 396-408. 10.1145/366663.366704. ACM.

[50] Unity Technologies. 2018. *Unity - Manual: Coroutines*. https://docs.unity3d.com/Manual/Coroutines.html.

[51] Harold Coopper. 2012-12. *Coroutine Event Loops in Javascript*. https://x.st/javascript-coroutines/.

[52] Kotlin. 2018-12-06. *Kotlin Documentation*. *Shared mutable state and concurrency.* GitHub. https://github.com/Kotlin/kotlinx.coroutines/blob/1.1.1/docs/shared-mutable-state-and-concurrency.md.

[53] Domenic Denicola. 2012-10-14. *You're Missing the Point of Promises*. https://blog.domenic.me/youre-missing-the-point-of-promises/.

[54] Simon Peyton Jones. Philip Wadler. 1992. 1993-01. *Imperative Functional Programming*. ACM. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. 71-84. 0-89791-560-7. 10.1145/158511.158524. https://www.microsoft.com/en-us/research/wp-content/uploads/1993/01/imperative.pdf.

[55] Paul Hudak. John Hughes. Simon Peyton Jones. Philip Wadler. 2007-04-16. *A History of Haskell: Being Lazy with Class*. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. 12-1--12-55. 978-1-59593-766-7. 10.1145/1238844.1238856. ACM. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf.

[56] The University of Glasgow. 2010. *base-4.12.0.0: Basic libraries*. *Control.Concurrent.* Hackage. http://hackage.haskell.org/package/base/docs/Control-Concurrent.html#v:forkIO.

[57] John A De Goes. 2017-09-16. *There Can Be Only One...IO Monad*. http://degoes.net/articles/only-one-io.

[58] Alexandru Nedelcu. 2018-11-09. *Task - Monix*. *Documentation.* GitHub. https://monix.io/docs/3x/eval/task.html.

[59] Viktor Klang. Lightbend, Inc.. 2017-12-19. *Reactive Streams*. GitHub. http://www.reactive-streams.org/.

[60] Mozilla. 2018-09-23T04:04:54Z. *JavaScript - Concurrency model and Event Loop*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[61] Adam Retter. *xq-promise Terminology vs. JavaScript/jQuery*. 2018-11-30. GitHub. https://github.com/james-jw/xq-promise/issues/19.

[62] Adam Retter. 2019-01-13. *Haskell I/O and XPath*. https://blog.adamretter.org.uk/haskell-io-and-xpath/.

[63] Giorgio Ghelli. Christopher Ré. Jérôme Siméon. 2006. *XQuery!: An XML query language with side effects*. *Current Trends in Database Technology -- EDBT 2006*. Springer Berlin Heidelberg. 178-191. 978-3-540-46790-8.

[64] Saxonica. 2018-12-06. *Saxon Documentation*. *Tuple types.* Saxonica. http://www.saxonica.com/documentation/index.html#!extensions/syntax-extensions/tuple-types.

Ex-post rule match selection: A novel approach to XSLT-based Schematron validation

David Maus
Herzog August Bibliothek Wolfenbüttel
<maus@hab.de>

Abstract

SchXslt [6] is a Schematron processor written entirely in XSLT. It follows the principal design of Rick Jelliffe's "skeleton" implementation and compiles a Schematron 2016 schema to a validating XSLT stylesheet. The goal of the SchXslt project is a conforming processor that improves Schematron validation by using features and instructions of recent XSLT versions. This paper discusses the principal design of an XSLT-based Schematron processor and introduces ex-post rule match selection as a novel validation strategy.

1. Introduction

Schematron is a rule based validation language for structured documents. It was designed by Rick Jelliffe in 1999 and standardized as ISO/IEC 19757-3 in 2006. The key concepts of Schematron validation are *patterns* that are the focus of a validation, *rules* selecting the portions of a document contributing to the pattern, and *assertion tests* that are run in the context of a rule. Schematron uses XPath both as the language to select the portion of a document and as the language of the assertion tests. This use of XPath gives Schematron the flexibility to validate arbitrary relationships and dependencies of information items in a document.

What also sets Schematron apart from other languages is that it encourages the use of natural language descriptions targeted to human readers. This way validation can be more than just a binary distinction (document valid/invalid) but also support authors of in-progress documents with quick feedback on erroneous or unwanted document structure and content.

2. Design of an XSLT-based Schematron processor

The principal design of an XSLT-based Schematron processor was laid out as early as 1999 by [2] and [4], later summarized by [1]. An XSLT-based processor reads a Schematron document and transforms it into an XSLT stylesheet. This *val-*

idation stylesheet is then applied to an XML document and outputs a *validation report*.

Given the key concepts of the Schematron language one can describe the basic structure of a validation stylesheet:

- A *pattern* is implemented as a named *pattern template*.
- A *rule* is implemented as a *rule template* with the rule context expression as match expression. Rule templates are chained by calls to `xsl:apply-templates`.
- An *assertion tests* is implemented as `xsl:if` element with the assertion in the `test` attribute.

The standardization of Schematron ([5]) added two concepts to earlier versions of Schematron that make compiling a validation stylesheet a three-stage process: Abstract patterns and rules, and external definitions.

- A pattern can be declared as abstract and use named parameters that act as placeholders and are replaced when the abstract pattern is instantiated. An abstract rule is a collection of assertions and reports without a rule context expression.
- External definitions enable sharing patterns, rules, or assertion tests between Schematron files.

The three step compilation works as follows. The first step copies the external definitions in the source document. The second step instantiates abstract patterns and rules. The final third step transforms the resulting Schematron into the validation stylesheet. Once this stylesheet is compiled it is applied to an XML document and creates a validation report using the Schematron Validation Report Language (SVRL).

The relationship of a pattern and its rules is represented by a pattern specific XSLT *mode*. Because the match selection of an `xsl:apply-templates` instruction selects only one template per node according to a template's priority and import precedence, a node N mached by rule R1 in Pattern P1 would not be matched by a rule R2 in Pattern P2 unless the rule templates run in different modes. To cover the case where two rules of the same pattern inadvertently match the same node all rule templates are created with a calculated priority reflecting their relative position.

Example 1 shows a simple Schematron with two patterns (P1, P2) and five rules (R1, R2, R3, R4, R5). To simplify the example each rule has exactly one assertion tests that always fails. Example 2 shows a corresponding validation stylesheet. Figure 1 visualizes the validation process. The rules R1, R2 and R3, R4 are chained by calls to `xsl:apply-templates` in two modes M1 and M2. The rule R5 is never tested because it matches the same context as R4 and has a lower priority.

Modified default template rules ensure that every node of the XML document can be matched by a rule template.

Example 1. Simple Schematron

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
        queryBinding="xslt2">
  <pattern id="P1">
    <rule context="/" id="R1">
      <assert test="false()" id="A1"/>
    </rule>
    <rule context="*" id="R2">
      <assert test="false()" id="A2"/>
    </rule>
  </pattern>
  <pattern id="P2">
    <rule context="@attribute" id="R3">
      <assert test="false()" id="A3"/>
    </rule>
    <rule context="element" id="R4">
      <assert test="false()" id="A4"/>
    </rule>
    <rule context="element" id="R5">
      <assert test="false()" id="A5"/>
    </rule>
  </pattern>
</schema>
```

Example 2. Validation stylesheet

```
<xsl:transform version="2.0"
               xmlns:svrl="http://purl.oclc.org/dsdl/svrl"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output indent="yes"/>

  <xsl:template match="/">
    <svrl:schematron-output>
      <xsl:call-template name="P1"/>
      <xsl:call-template name="P2"/>
    </svrl:schematron-output>
  </xsl:template>

  <xsl:template name="P1">
    <svrl:active-pattern id="P1"/>
    <xsl:apply-templates mode="M1" select="."/>
  </xsl:template>
```

```
<xsl:template name="P2">
  <svrl:active-pattern id="P2"/>
  <xsl:apply-templates mode="M2" select="."/>
</xsl:template>

<xsl:template match="/" mode="M1" priority="1">
  <svrl:fired-rule id="R1"/>
  <xsl:if test="not(false())">
    <svrl:failed-assert id="A1"/>
  </xsl:if>
  <xsl:apply-templates select="node() | @*" mode="#current"/>
</xsl:template>

<xsl:template match="*" mode="M1" priority="0">
  <svrl:fired-rule id="R2"/>
  <xsl:if test="not(false())">
    <svrl:failed-assert id="A2"/>
  </xsl:if>
  <xsl:apply-templates select="node() | @*" mode="#current"/>
</xsl:template>

<xsl:template match="@attribute" mode="M2" priority="2">
  <svrl:fired-rule id="R3"/>
  <xsl:if test="not(false())">
    <svrl:failed-assert id="A3"/>
  </xsl:if>
  <xsl:apply-templates select="node() | @*" mode="#current"/>
</xsl:template>

<xsl:template match="element" mode="M2" priority="1">
  <svrl:fired-rule id="R4"/>
  <xsl:if test="not(false())">
    <svrl:failed-assert id="A4"/>
  </xsl:if>
  <xsl:apply-templates select="node() | @*" mode="#current"/>
</xsl:template>

<xsl:template match="element" mode="M2" priority="0">
  <svrl:fired-rule id="R5"/>
  <xsl:if test="not(false())">
    <svrl:failed-assert id="A5"/>
  </xsl:if>
  <xsl:apply-templates select="node() | @*" mode="#current"/>
</xsl:template>
```

```
  <xsl:template match="node() | @*" priority="-10" mode="#all">
    <xsl:apply-templates select="node() | @*" mode="#current"/>
  </xsl:template>

</xsl:transform>
```


Figure 1. Processing the Rules R1–R4 in two modes

3. Ex-post rule match selection

Using one mode per pattern as shown in example 2 is required to allow rules from different patterns to match the same node, but has a drawback: The entire source document is processed as many times as there are patterns in the Schematron.

SchXslt addresses this by implementing a different validation strategy, *ex-post rule match selection*. This strategy relies on the `xsl:next-match` instruction introduced with XSLT 2.0. When called, it applies the next template that matches the current context node but has a lower priority, or the built-in template rules if no other template matching the current context node is found ([3]). The `xsl:next-match` instruction overcomes a limitation of XSLT 1.0 where matching a node with more than one template was relegated to imported templates (`xsl:apply-imports`), or template modes. Second, ex-post rule match selection looks at the Schematron validation from the perspective of the validation report. From here there is no difference between a rule that never fired and a rule that fired but is not reported. Hence if we were to assume that the case where two rules of a pattern match the same node is an error in the Schematron, then we could see it as justified to fire such a rule but remove it from the report.

Ex-post rule match selection thus works as follows: The Schematron processor chains all rules with a call to `xsl:next-match` instead of `xsl:apply-templates`.

This will fire all rules of all patterns that match a node in the source document. The validation stylesheet collects this information in a temporary report and removes rules that fired because they matched a node that was already matched by a previous rule in the same pattern. To do so the validation stylesheet uses the generated id of the current context node and the generated id of the pattern during compilation to track which nodes have been matched by which rules in which patterns. To ensure that the order of reported fired rules reflects the order of rules in a pattern, the processor calculates a priority for each template such that a template which is lexically further down in the Schematron has a lower priority than one higher up.

Example 3 shows the simple Schematron implemented with ex-post rule match selection. Note how the identity of context and pattern are stored in a `schxslt:context` and `schxslt:pattern` attribute, and are later used to remove all but the first fired rule to be reported. Also note how the generated id for the pattern was created during the compilation phase and the one for the context node is generated by the validation stylesheet. Figure 2 shows how the Rules R2, R4, and R5 are chained by `xsl:next-match` with the match of R5 later to be removed.

Example 3. Validation stylesheet using ex-post rule match selection

```
<xsl:transform version="2.0"
               xmlns:schxslt="https://doi.org/10.5281/zenodo.1495494"
               xmlns:svrl="http://purl.oclc.org/dsdl/svrl"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output indent="yes"/>

  <xsl:template match="/">
    <xsl:variable name="report">
      <xsl:call-template name="Validate"/>
    </xsl:variable>
    <svrl:schematron-output>
      <xsl:for-each select="$report/svrl:active-pattern">
        <xsl:copy>
          <xsl:sequence select="@* except @schxslt:*"/>
        </xsl:copy>
        <xsl:for-each-group select="$report/svrl:fired-
rule[@schxslt:pattern = current()/@schxslt:pattern]"
                            group-by="@schxslt:context">
          <xsl:copy>
            <xsl:sequence select="@* except @schxslt:*"/>
          </xsl:copy>
          <xsl:sequence select="*"/>
        </xsl:for-each-group>
```

```
      </xsl:for-each>
    </svrl:schematron-output>
  </xsl:template>

  <xsl:template name="Validate">
    <svrl:active-pattern id="P1" schxslt:pattern="d11e-1"/>
    <svrl:active-pattern id="P2" schxslt:pattern="d11e-2"/>
    <xsl:apply-templates mode="M" select="."/>
  </xsl:template>

  <xsl:template match="/" mode="M" priority="4">
    <svrl:fired-rule id="R1" schxslt:pattern="d11e-1"
schxslt:context="{generate-id()}">
      <xsl:if test="not(false())">
        <svrl:failed-assert id="A1"/>
      </xsl:if>
    </svrl:fired-rule>
    <xsl:next-match/>
  </xsl:template>

  <xsl:template match="*" mode="M" priority="3">
    <svrl:fired-rule id="R2" schxslt:pattern="d11e-1"
schxslt:context="{generate-id()}">
      <xsl:if test="not(false())">
        <svrl:failed-assert id="A2"/>
      </xsl:if>
    </svrl:fired-rule>
    <xsl:next-match/>
  </xsl:template>

  <xsl:template match="@attribute" mode="M" priority="2">
    <svrl:fired-rule id="R3" schxslt:pattern="d11e-2"
schxslt:context="{generate-id()}">
      <xsl:if test="not(false())">
        <svrl:failed-assert id="A3"/>
      </xsl:if>
    </svrl:fired-rule>
    <xsl:next-match/>
  </xsl:template>

  <xsl:template match="element" mode="M" priority="1">
    <svrl:fired-rule id="R4" schxslt:pattern="d11e-2"
schxslt:context="{generate-id()}">
      <xsl:if test="not(false())">
        <svrl:failed-assert id="A4"/>
      </xsl:if>
```

```
    </svrl:fired-rule>
    <xsl:next-match/>
  </xsl:template>

  <xsl:template match="element" mode="M" priority="0">
    <svrl:fired-rule id="R5" schxslt:pattern="d11e-2"
schxslt:context="{generate-id()}">
      <xsl:if test="not(false())">
        <svrl:failed-assert id="A5"/>
      </xsl:if>
    </svrl:fired-rule>
    <xsl:next-match/>
  </xsl:template>

  <xsl:template match="node() | @*" priority="-10" mode="#all">
    <xsl:apply-templates select="node() | @*" mode="#current"/>
  </xsl:template>

</xsl:transform>
```

Figure 2. Processing the Rules R1–R5 with ex-post rule match selection

The Schematron given in example 1 sure is a simplification of Schematron's feature set. There are two features in particular that question the applicability of the `xsl:next-match` instruction.

- Schematron supports variable bindings in the scope of a pattern. These variables can be used in XPath expressions inside rules. Rules that use pattern scoped variable bindings can only run together if they use the same variable bindings.
- A pattern can specify documents other than the source document to be validated by its rules. Rules can only run together if they apply to the same document.

SchXslt takes this into account and identifies patterns whose rules can run together in the same mode because they validate the same documents. Patterns

with pattern scoped variable bindings each run in a single mode. The grouping is implemented by a grouping key function that concatenates the generated id of a pattern's variable binding element (`sch:let`) and the value of a patterns `documents` attribute. This way ex-post rule match selection reduces the number of times the source documents are processed to one in the best case and to be equal to the number of patterns in the worst case.

4. Conclusion and future work

SchXslt is a conforming XSLT-based Schematron processor. It implements Schematron validation with novel strategy that groups patterns whose rules can run at once and removes unwanted report elements afterwards. Up to now the development of SchXslt was focussed on implementing a conforming processor and exploring recent XSLT features like the `xsl:next-match` instruction.

The goal for the near future is to use SchXslt in real-world validation scenarios and see under which circumstances ex-post rule match selection improves the overall validation performance.

Bibliography

[1] Dodds, Leigh: *Schematron: validating XML using XSLT*. XSLT-UK conference, April 8-9 2001 `http://ldodds.com/papers/schematron_xsltuk.html`

[2] Jelliffe, Rick: *Using XSL as a Validation Language*. Internet Document, 1999 `https://web.archive.org/web/20000415135808/http://www.ascc.net:80/xml/en/utf-8/XSLvalidation.html`

[3] Kay, Michael (ed.): *XSL Transformations (XSLT) Version 2.0 (Second Edition)*. W3C Proposed Edited Recommendation, 21 April 2009 `http://www.w3.org/TR/xslt20/`

[4] Norton, Francis: *Generating XSL for Schema Validation*. Internet Document, 1999 `https://web.archive.org/web/20010110235200/http://www.redrice.com/ci/generatingXslValidators.html`

[5] *Information Technology – Document Schema Definition Languages (DSDL) – Part 3: Rule-based validation – Schematron*. 2016 (=ISO/IEC-19757-3:2016)

[6] Maus, David: *SchXslt*. `DOI: 10.5281/zenodo.1495494`[1]

[1] https://doi.org/10.5281/zenodo.1495494

Authoring Domain Specific Languages in Spreadsheets Using XML Technologies

Alan Painter
HSBC France
<alan.painter@hsbc.fr>

Abstract

Domain Specific Languages (DSLs) have been shown to be useful in the development of information systems. DSLs can be designed to be authored, manipulated and validated by business experts (BEs) and subject matter experts (SMEs). Because BEs and SMEs are known to be comfortable working with spreadsheet applications (Microsoft™ Excel®, Libre Office Calc), the ability to author DSLs within spreadsheets makes the DSL authoring process even more engaging for BEs and SMEs.

Today's most popular spreadsheet applications are implemented using XML documents (ODF, OOXML, Excel 2003 XML format) and, for this reason at least, XML technologies (XPATH, XSLT, XQuery) are well suited for reading DSL definitions within Spreadsheets.

What is usually considered the part of DSL implementation that requires the most effort is the artifact or code generation from the DSL description. For this aspect of DSL development, XML technologies are also well placed for generating the technical artifacts described by the DSLs.

In this paper, I will first motivate the usage of DSLs by describing some of their utility in information systems development. I'll then go on to describe how XML Technologies can be used for reading DSLs within spreadsheets and for generating technical artifacts. I'll then go on to present some real world examples of DSL usage via XML Technologies and attempt to draw some general conclusions from the examples.

Keywords: XML, DSL, Domain Specific Languagues, Spreadsheet

1. An Entirely Incomplete Description of DSLs

Domain Specific Languages (DSLs) are not a new concept in Information Technology[1]. The examples are legion and familiar to developers. To cite just a few well-known examples: *make* [2], *YACC* [3], *troff* [4], *html* [5]. These are computer languages that are used by humans in order to describe how to produce an artifact or present a graphical user interface.

Table 1. Well Known Examples of DSLs

Example	Model	Generated Artifact	Method
make	Dependency Graph between source files and generated files	Any generated artifact (often generated libraries and executables)	Runs system commands to launch compilers, applications, file manipulation, etc.
YACC	Grammar describing a computer language.	*C*-language source code for recognizing and processing the language described in the grammar.	Directly generated by *YACC*.
troff	Text processing	Device-independent output format	Directly generated by *troff*
html	Text, table, form markup and linkages between hypermedia documents (simplified description)	User Interface of Rendered Views and Embedded Controls	Browser reads the html, retrieves associated documents, renders and presents the UI.

The examples above underline the two major objectives of a DSL to consider:

1. the syntax of the language itself which allows expressing the different elements, their attributes and relations within a specific domain
2. the implementation of the language which produces the resulting artifacts described by the input

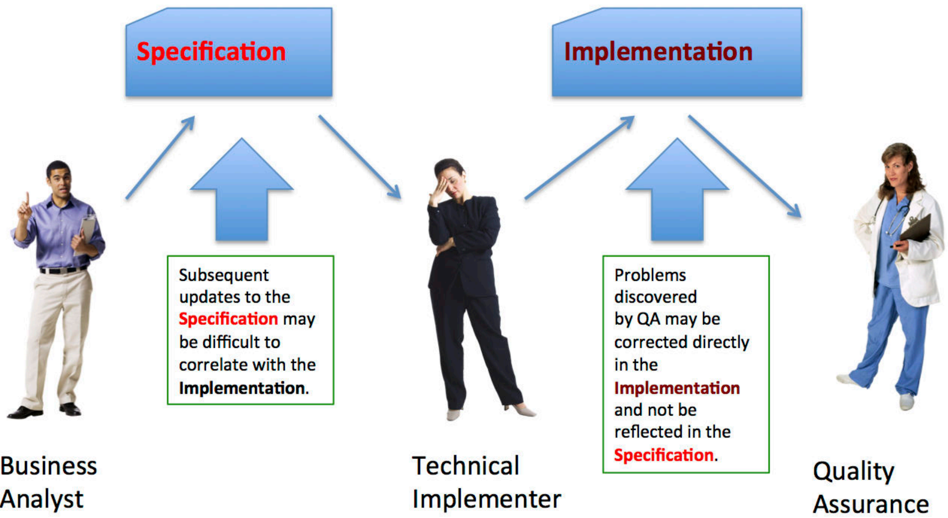
2. DSLs in Business Application Development

There has been a recent resurgence in interest for using DSLs for describing rules and processes for business applications. [6] [7] Martin Fowler dedicated a book to the subject of DSLs [8] and mentions:

> *I believe that the hardest part of software projects, the most common source of project failure, is communication with the customers and users of that software. By providing a clear yet precise language to deal with domains, a DSL can help improve this communication.*
>
> —*Martin Fowler[8]*

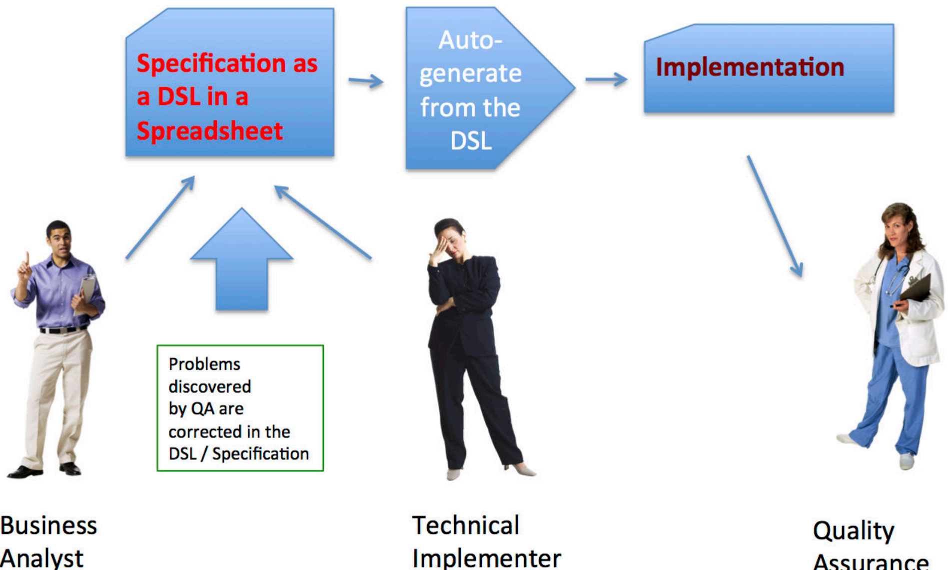
Business software developers are often faced with the challenge of working with different Domain Experts (DEs), Subject Matter Experts (SMEs) and Business

Experts (BEs) who are involved in the specification and definition of what the business application needs to do, often as part of the business requirements or functional requirements specification. Technical developers will then take the specification and interpret it by developing the technical implementation. At this point, we have two separate representations of the work: the specification and the implementation.



Because there are two separate representations, these two can evolve separately and differently and, subsequently, diverge. The informal specification may no longer be up-to-date because the technical implementation may have corrected problems discovered either during technical analysis or by quality assurance. Updates made to the informal specification may not be obvious to add to the technical implementation based upon a previous version of the informal specification.

If the Business Analysts and Technical Developers can agree upon a DSL for representing the business rules and process description, they can then have a chain of development in which the Business Analyst becomes a contributor to the DSL specification. The Technical Implementor can also contribute to the DSL specification. The DSL specification is then the common support for the business implementation from which the implementation of the business rules will be generated automatically from the DSL specification. This means that the specification (the DSL) is alway up-to-date and synchronized with the technical implementation (generated artifact).



A further advantage of providing a DSL to the Business Analyst is that this allows the Business Analyst the possibility of testing the business rules immediately, given an appropriate test framework. This means that the Business Analyst can contribute to the testing of the new rules and can verify the specification immediately rather than awaiting its implementation.

In this paper, I'll address what I see as the two major advantages of using a DSL for application development.

- The representation of the functional workings afforded by a DSL, separate from the technical corpus.
- The optimization of the development process that is gained by enabling technical and non-technical contributors to author the functional definition via a DSL.

3. DSLs in Spreadsheets

Given that DSLs are useful for engaging Business Experts in the specification, development and testing process, it will also be useful to allow the Business Experts to edit the DSL within a Spreadsheet application. Spreadsheet applications such as Microsoft Excel and LibreOffice Calc are common tools for Business Experts to use and the comfort level with using spreadsheet applications is high.

There are several aspects of spreadsheet editing that make it a favorite among Business Experts:

- Spreadsheets allow for realigning and reordering columns and lines in simple gestures
- Orthogonal editing facilitate keeping lists in one axis and attributes of each item in the list in the other

- Spreadsheet applications allow for defining styles on a cell-by-cell basis (text color, fonts, background colors) and sometimes even within the text of the cell

Not all DSLs will be obvious to author from within a spreadsheet; nonetheless, there will be a subset of DSLs that can be represented in tabular form. What follows in this paper will address that subset of DSLs that can be authored from within a spreadsheet application.

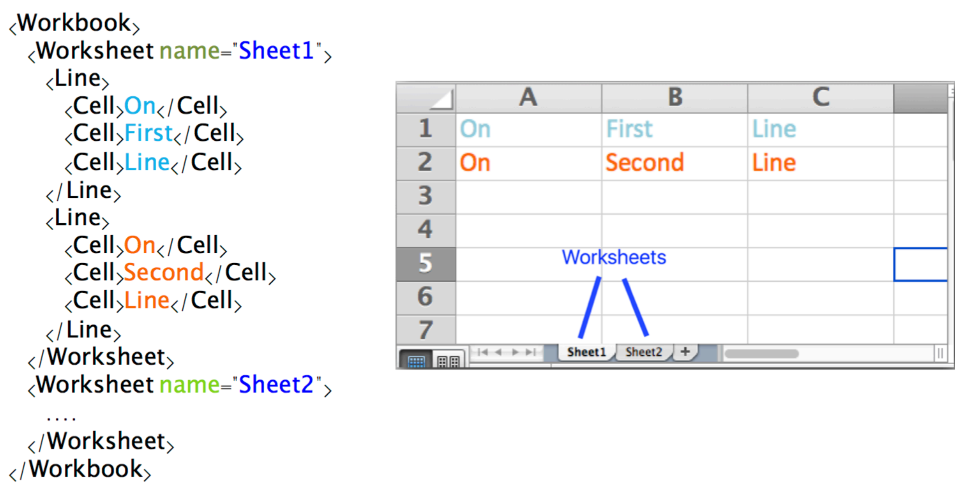
4. Using XML Technologies To Read Spreadsheet Data

4.1. The Simple Structure of Data Within a Spreadsheet

Treating spreadsheets as tables of data [1] gives a simple model of the spreadsheet structure in XML with four nested elements:

- Workbook
- Worksheet (with a required *name* attribute)
- Line
- Cell

The following image gives an example of the simple model and its relationship to a spreadsheet.



4.2. Using XPATH for Addressing Data within the Simple Model

The data within the Simple Model is easily addressed using XPATH. For example, data can be addressed using fixed coordinates. In the following, the second *Cell* element of the first *Line* is chosen.

[1]Ignoring formulas, formatting and other non-data spreadsheet notations.

```
/Workbook/Worksheet[@name eq 'Sheet1']/Line[1]/Cell[2]
```

Data can be addressed based upon its content. The following selects all of the *Cell* elements in the second line of the first *Worksheet* because that line's second *Cell* contains the value *'Second'*

```
/Workbook/Worksheet[@name eq 'Sheet1']/Line[Cell[2] eq 'Second']/
Cell[3]
```

The simplicity of addressing data within the simple spreadsheet model will aid in reading the DSL contained within the spreadsheet.

4.3. Extracting the Simple Spreadsheet Model from Real-world Spreadsheets

Extracting the Simple Spreadsheet Model from the different real-world spreadsheet vocabularies is a bit more complicated and it is somewhat beyond the scope of this paper.

Nonetheless, to give an idea of the complexity, I would mention that I know of an implementation for reading *.xslx* files in *"strict"* mode that requires around 250 lines of XSLT. Another implementation for Excel 2003 XML mode requires fewer than 100 lines of XSLT. Hence, the problem of extracting the Simple Spreadsheet Model from a spreadsheet is fairly simple to achieve.

5. Using XML Technology to Generate Artifacts from a DSL

I'm concentrating on XSLT in the following examples although I believe that XQuery would hold as well. Standard XSLT does, nonetheless, have the advantage over Standard XQuery of facilitating output to multiple documents.

5.1. Producing artifacts using TEXT output

XSLT writes in a "text" output method as a series of "string" data. This is sufficient, although not necessarily convenient, for writing output such as GPL languages (Java, C, C++) or for writing configuration files such as Java property files, Windows .ini files, etc.

The *Text Value Template* mechanism in XSLT3 can be particularly useful and elegant for producing text. [2]

[2]https://www.w3.org/TR/xslt-30/#text-value-templates

5.2. Producing XML and JSON artifacts

Writing XML documents is, of course, the natural output method for XSLT, hence XSLT is the ideal language for such output. JSON output is also a possibility with XSLT, especially in the latest version, XSLT3.0, which has a JSON output method.

5.3. Producing XSLT artifacts

XSLT is a special form of XML output but it is significant to mention that XSLT is very good at writing XSLT.

Where XSLT is particularly useful is that it is a separable artifact that can be tested independently and then be included into a larger, technical project. In essence, this allows us to generate the XSLT from the DSL and test the results separately, especially by the Business Analyst but also within a unit testing framework. The XSLT itself can then be included, at compile-time or run-time, into a separate technical corpus, especially Java but also with C#, C, C++, Javascript, JVM-based languages such as scala, etc.

The process of using `xsl:namespace-alias` and `xsl:element` in order to facilitate the generation of XSLT artifacts from another XSLT is described in the XSLT literature.[9]

6. Some examples of DSLs in Spreadsheets

To motivate the utility and convenience of developing DSLs within Spreadsheets using XML Technologies, I'll give a few examples from my personal work experience. These are examples that I have used extensively and which have provided an important benefit to different projects.

Table 2. Some Personal Examples of DSLs in Spreadsheets

<table>
<tr><th>Use Case</th><th>DSL Model</th><th>Output Artifact</th><th>Output Method</th></tr>
<tr><td rowspan="2">Automaton</td><td rowspan="2">Table of States, Events, Transitions and Actions</td><td>Java Abstract Class</td><td rowspan="2">text</td></tr>
<tr><td>Graphviz DOT language</td></tr>
<tr><td>Configuration</td><td>Linked Description of Instances and Values
Templates of Properties</td><td>Properties files
XML Configuration
JSON Configuration
YML Configuration</td><td>text
xml
json</td></tr>
</table>

Use Case	DSL Model	Output Artifact	Output Method
Tabular Data from XML	for-each / variables / cell rules	XSLT	xml
Schema to Schema	templates, variables, transcodification	XSLT	xml

6.1. An Automaton or Finite State Machine (FSM) as DSL

6.1.1. Objectives of the FSM DSL in a Spreadsheet

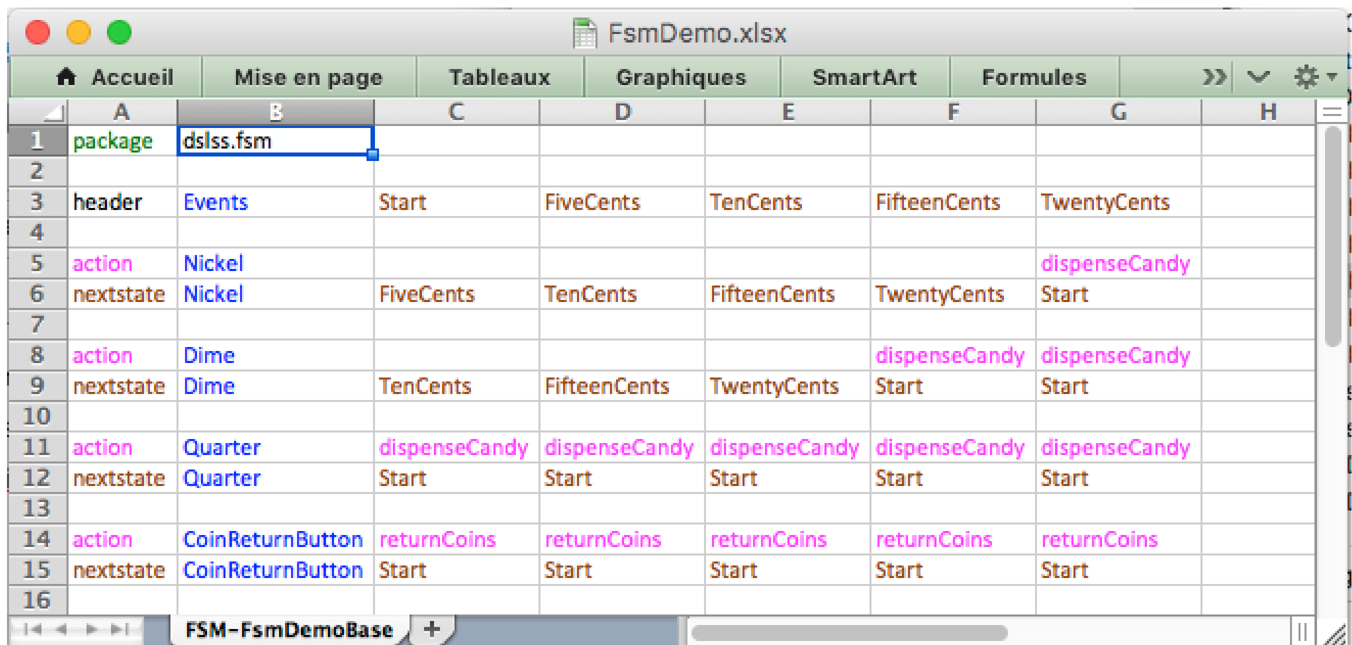
- Make it easy to edit the standard tabular represention of an FSM with its list of *States*, *Events*, *Actions* and *Transitions*.
- Generate a Java class that implements the *handleEvent()* method
- Generate a *DOT* file description of the FSM from which GraphViz will generate the state/transition graph

6.1.2. Description of the DSL Model

A common and basic representation in computer science is to show a Finite State Machine in tabular form with *States* represented in columns and *Events* represented in rows.

The given example is that of a coin vending machine in the USA with the *Events* being named after the US coins that can be deposited in the Vending Machine (i.e. a *Nickel* is 5 cents, a *Dime* is 10 cents and a *Quarter* is 25 cents. An additional *Event* is the *CoinReturnButton*.. The *States* are the cumulative total of coins dropped in the machine up to 25 cents, at which point the *candy* is dispensed and the state returns to *Start*. [3]

[3]Note that this is a very simple machine and that change is not rendered for amounts over 25 cents.



	A	B	C	D	E	F	G	H
1	package	dslss.fsm						
2								
3	header	Events	Start	FiveCents	TenCents	FifteenCents	TwentyCents	
4								
5	action	Nickel					dispenseCandy	
6	nextstate	Nickel	FiveCents	TenCents	FifteenCents	TwentyCents	Start	
7								
8	action	Dime				dispenseCandy	dispenseCandy	
9	nextstate	Dime	TenCents	FifteenCents	TwentyCents	Start	Start	
10								
11	action	Quarter	dispenseCandy	dispenseCandy	dispenseCandy	dispenseCandy	dispenseCandy	
12	nextstate	Quarter	Start	Start	Start	Start	Start	
13								
14	action	CoinReturnButton	returnCoins	returnCoins	returnCoins	returnCoins	returnCoins	
15	nextstate	CoinReturnButton	Start	Start	Start	Start	Start	
16								

Figure 1. A Representation of an Automaton with a Spreadsheet

This tabular form in a spreadsheet simplifies the authoring. Adding a new state means adding a new column. Adding a new event is adding two new rows (one row for the *action* and one row for the *nextstate*. The author can copy/paste a cell in order to duplicate an existing *action* or *nextstate*. Both *states* transitions and *actions* are captured in this simple table.

The simple worksheet model in XML looks something like this (abbreviated):

```
<Workbook>
    <Worksheet name="FSM-FsmDemoBase">

        <Line>
            <Cell>package</Cell><Cell>dslss.fsm</Cell>
        </Line>

        <Line>
            <Cell>header</Cell><Cell>Events</Cell><Cell>Start</
Cell><Cell>FiveCents</Cell><Cell>TenCents</Cell><Cell>FifteenCents</
Cell><Cell>TwentyCents</Cell>
        </Line>

        <Line>
            <Cell>action</Cell><Cell>Nickel</Cell><Cell></Cell><Cell></
Cell><Cell></Cell><Cell></Cell><Cell>dispenseCandy</Cell>
        </Line>
        <Line>
            <Cell>nextstate</Cell><Cell>Nickel</Cell><Cell>FiveCents</
Cell><Cell>TenCents</Cell><Cell>FifteenCents</Cell><Cell>TwentyCents</
```

```
Cell><Cell>Start</Cell>
        </Line>

        <Line>
            <Cell>action</Cell><Cell>Dime</Cell><Cell></Cell><Cell></
Cell><Cell></Cell><Cell>dispenseCandy</Cell><Cell>dispenseCandy</Cell>
        </Line>
        <Line>
            <Cell>nextstate</Cell><Cell>Dime</Cell><Cell>TenCents</
Cell><Cell>FifteenCents</Cell><Cell>TwentyCents</Cell><Cell>Start</
Cell><Cell>Start</Cell>
        </Line>

        ....

    </Worksheet>
</Workbook>
```

6.1.3. Describing the Java Abstract Class

One artifact that the DSL implementation is to generate is a *java abstract class* that encapsulates the generic *handleEvent*() method and the *state/action* tables for the FSM taken from the spreadsheet DSL. The FSM table gives us the names of the *action* methods and hence the generated class declares these methods as *abstract* in such a fashion that the deriving, concrete class can then simply implement the action methods.

In our simple example, the generated Java abstract class can look like:

```
package dslss.fsm;

public abstract class FsmDemoBase {

    public enum Event { Nickel, Dime, Quarter, CoinReturnButton }

    public enum State { Start, FiveCents, TenCents, FifteenCents,
TwentyCents }

    protected abstract Runnable dispenseCandy();
    protected abstract Runnable returnCoins();

    private final Runnable action[][] = {
        { __nop(),          __nop(),          __nop(),
__nop(),          dispenseCandy() },
        { __nop(),          __nop(),          __nop(),
dispenseCandy(), dispenseCandy() },
```

```
        { dispenseCandy(), dispenseCandy(), dispenseCandy(),
dispenseCandy(), dispenseCandy() },
        { returnCoins(),   returnCoins(),   returnCoins(),
returnCoins(),   returnCoins()   },
    };

    private final static State nextState[][] = {
        { State.FiveCents, State.TenCents,     State.FifteenCents,
State.TwentyCents, State.Start },
        { State.TenCents,  State.FifteenCents, State.TwentyCents,
State.Start,       State.Start },
        { State.Start,     State.Start,        State.TwentyCents,
State.Start,       State.Start },
        { State.Start,     State.Start,        State.TwentyCents,
State.Start,       State.Start },
    };

    public final State handleEvent(final State currentState, final Event
newEvent) {
                action [newEvent.ordinal()]
[currentState.ordinal()].run();
        return nextState [newEvent.ordinal()] [currentState.ordinal()];
    }

    private Runnable __nop() { return () -> {}; }
}
```

and, for reference, an example concrete Java class that extends the generated abstract class can be:

```
package dslss.fsm;

public class FsmDemo extends FsmDemoBase {

    protected Runnable dispenseCandy() {
        return () -> System.out.println("Dispensing candy.");
    }

    protected Runnable returnCoins() {
        return () -> System.out.println("Returning coins.");
    }
}
```

6.1.4. A Stylesheet for Generating the Java Abstract Class

To generate the java abstract class from Simple Spreadsheet Model using XML Technologies, I'll give an example using *XSLT*. The template will match any *Worksheet* element with a name that starts with the substring *FSM*.

```
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
                xmlns:xs  = "http://www.w3.org/2001/XMLSchema"
                xmlns:f   = "urn:for:functions"

                version="3.0" >

    <xsl:param name="outputDir" as="xs:string" />

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

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

    <xsl:template match="Worksheet[starts-with(@name, 'FSM-')]" expand-
text="yes" >
    ...
```

and within that template we'll have the static Java code that we'll generate as well as different *Text Value Template* (TVT) fields that we'll use to produce the specific *Event* names, *State* names, *Action* method declarations and the values for the *action* and *nextstate* tables. The result is fairly readable.

```
    <xsl:template match="Worksheet[starts-with(@name, 'FSM-')]" expand-
text="yes" >

        <xsl:variable name="class" as="xs:string" select="substring-
after(@name, 'FSM-')" />

        <xsl:result-document href="{$outputDir}/{f:getPath(Line)}/
{$class}.java" method="text">
package {f:getPackage(Line)}

public abstract class {$class} {{

    public enum Event {{ {f:getEventsList(Line)} }}

    public enum State {{ {f:getStatesList(Line)} }}
```

```
{f:getMethodDeclarationLines(Line)}

    private final Runnable action[][] = {{
{f:getFormattedTable(f:getActionTable(Line))}
     }};

    private final static State nextState[][] = {{
{f:getFormattedTable(f:getNextStateTable(Line))}
     }};

    public final State handleEvent(final State currentState, final Event
newEvent) {{
                  action [newEvent.ordinal()]
[currentState.ordinal()].run();
        return nextState [newEvent.ordinal()] [currentState.ordinal()];
    }}

    private Runnable __nop() {{ return () -> {{}}; }}
}}
        </xsl:result-document>
    </xsl:template>
```

The functions for extracting the *action* and *state* names for the Java *enum* declarations are especially simple:

```
    <xsl:function name="f:getPackage" as="xs:string">
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:sequence select="$lines[Cell[1] eq 'package' ]/Cell[2]" />
    </xsl:function>

    <xsl:function name="f:getPath" as="xs:string">
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:sequence select="replace(f:getPackage($lines), '[.]',
'/')" />
    </xsl:function>

    <xsl:function name="f:getStates" as="xs:string*" >
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:sequence select="$lines[Cell[1] eq 'header']/
Cell[position() gt 2]" />
    </xsl:function>

    <xsl:function name="f:getStatesList" as="xs:string">
        <xsl:param name="lines" as="element(Line)*" />
```

```
        <xsl:value-of select="f:getStates($lines)" separator=", "/>
    </xsl:function>

    <xsl:function name="f:getEvents" as="xs:string*" >
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:sequence select="$lines[Cell[1] eq 'action']/Cell[2]" />
    </xsl:function>

    <xsl:function name="f:getEventsList" as="xs:string">
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:value-of select="f:getEvents($lines)" separator=", "/>
    </xsl:function>
```

The functions for creating the *state* and *action* tables and the abstract method declarations can also be fairly straightforward:

```
    <xsl:function name="f:getMethodDeclarationLines" as="xs:string">
        <xsl:param name="lines" as="element(Line)*" />

        <xsl:variable name="actions" as="xs:string*"
                      select="$lines[Cell[1] eq 'action']/
  Cell[position() gt 2]" />
        <xsl:variable name="methodDeclarations" as="xs:string*" expand-
  text="yes" >
            <xsl:for-each select="distinct-values($actions[. ne ''])" >
                <xsl:text>    protected abstract Runnable {.}();</
  xsl:text>
            </xsl:for-each>
        </xsl:variable>
        <xsl:value-of select="$methodDeclarations" separator="&#xA;" />
    </xsl:function>

    <xsl:function name="f:getActionTable" as="element(Line)*" >
        <xsl:param name="lines" as="element(Line)*" />

        <xsl:for-each select="$lines[Cell[1] eq 'action']">
            <Line>
                <xsl:for-each select="Cell[position() gt 2]" >
                    <Cell>
                        <xsl:value-of select="(text()[. ne ''], '__nop')
  [1] || '()'" />
                    </Cell>
                </xsl:for-each>
            </Line>
        </xsl:for-each>
```

```
    </xsl:function>

    <xsl:function name="f:getNextStateTable" as="element(Line)*" >
        <xsl:param name="lines" as="element(Line)*" />

        <xsl:for-each select="$lines[Cell[1] eq 'nextstate']">
            <Line>
                <xsl:for-each select="Cell[position() gt 2]" >
                    <Cell>
                        <xsl:value-of select="'State.' || text()" />
                    </Cell>
                </xsl:for-each>
            </Line>
        </xsl:for-each>
    </xsl:function>
```

6.1.5. Generating the DOT / GraphViz artifact

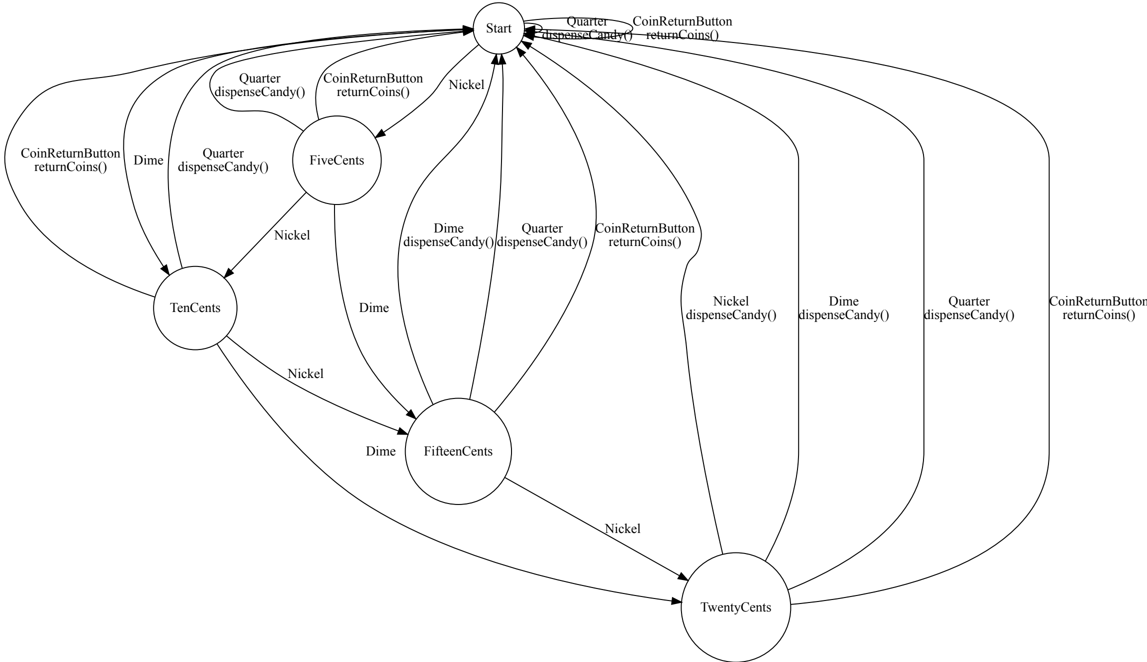
The *DOT*[11] file that we generate for the *graphviz* application contains a link from each of the states to a next state, labeled by the event that caused that state transition and a possible action. This is a simple text format for *DOT*

```
digraph FsmDemoBase {
    node [shape = circle];
    Start        -> FiveCents    [ label =
"Nickel"                          ];
    Start        -> TenCents     [ label =
"Dime"                            ];
    Start        -> Start        [ label = "Quarter
\ndispenseCandy()"        ];
    Start        -> Start        [ label = "CoinReturnButton
\nreturnCoins()" ];
    FiveCents    -> TenCents     [ label =
"Nickel"                          ];
    FiveCents    -> FifteenCents [ label =
"Dime"                            ];
    FiveCents    -> Start        [ label = "Quarter
\ndispenseCandy()"        ];
    FiveCents    -> Start        [ label = "CoinReturnButton
\nreturnCoins()" ];
    TenCents     -> FifteenCents [ label =
"Nickel"                          ];
```

```
    TenCents     -> TwentyCents  [ label =
"Dime"                            ];
    TenCents     -> Start        [ label = "Quarter
\ndispenseCandy()"        ];
    TenCents     -> Start        [ label = "CoinReturnButton
\nreturnCoins()" ];
    FifteenCents -> TwentyCents  [ label =
"Nickel"                          ];
    FifteenCents -> Start        [ label = "Dime
\ndispenseCandy()"           ];
    FifteenCents -> Start        [ label = "Quarter
\ndispenseCandy()"        ];
    FifteenCents -> Start        [ label = "CoinReturnButton
\nreturnCoins()" ];
    TwentyCents  -> Start        [ label = "Nickel
\ndispenseCandy()"         ];
    TwentyCents  -> Start        [ label = "Dime
\ndispenseCandy()"           ];
    TwentyCents  -> Start        [ label = "Quarter
\ndispenseCandy()"        ];
    TwentyCents  -> Start        [ label = "CoinReturnButton
\nreturnCoins()" ];
}
```

We can pass this document to the *dot* application in order to generate a graphic state presentation. In the following command, we will generate *SVG* output, but it's also possible to generate other graphic output types via *dot*.

```
% dot -Tsvg FsmDemoBase.gv -o FsmDemoBase.gv.svg
```



6.1.6. Generating the DOT artifact

We can use the *Text Value Template* mechanism again in order to generate this artifact, adding an extra output document to the same template that generated the *java abstract class* artifact above.

```
<xsl:result-document href="{$outputDir}/{f:getPath(Line)}/{$class}.gv"
method="text">
digraph {$class} {{
    node [shape = circle];
{f:getDotTable(f:getDotLines(Line))}
}}
</xsl:result-document>
```

To generate the *DOT* document, we need to create a link from each *state* to its *nextstate* and label it with the *event* that provoked the transaction, along with an *action* that may be triggered by the state transition. An example generation, again employing text value templates, could be:

```
    <xsl:function name="f:getDotLines" as="element(Line)*" expand-
text="yes">
        <xsl:param name="lines" as="element(Line)*" />

        <xsl:variable name="states" as="xs:string*"
select="f:getStates($lines)" />
        <xsl:for-each select="f:getStates($lines)" >
```

```
            <xsl:variable name="state" as="xs:string" select="." />
            <xsl:for-each select="f:getEvents($lines)" >
                <xsl:variable name="event" as="xs:string" select="." />
                <xsl:variable name="action" as="xs:string?"
select="f:getAction($lines, $state, $event)" />
                <Line>
                    <Cell>{$state}</Cell>
                    <Cell>-></Cell>
                    <Cell>{f:getNextState($lines, $state, $event)}</Cell>
                    <Cell>[ label = "{$event}{('\n' || $action || '()')
[$action]}"</Cell>
                    <Cell>];</Cell>
                </Line>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:function>

    <xsl:function name="f:getNextState" as="xs:string" >
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:param name="state" as="xs:string"      />
        <xsl:param name="event" as="xs:string"      />

        <xsl:variable name="stateColumn" as="xs:integer"
select="f:getHeaderIndex($lines, $state)" />
        <xsl:sequence select="$lines[Cell[1] eq 'nextstate'][Cell[2] eq
$event]/Cell[$stateColumn]" />
    </xsl:function>

    <xsl:function name="f:getAction" as="xs:string?" >
        <xsl:param name="lines" as="element(Line)*" />
        <xsl:param name="state" as="xs:string"      />
        <xsl:param name="event" as="xs:string"      />

        <xsl:variable name="stateColumn" as="xs:integer"
select="f:getHeaderIndex($lines, $state)" />
        <xsl:sequence select="$lines[Cell[1] eq 'action'][Cell[2] eq
$event]/Cell[$stateColumn]" />
    </xsl:function>
```

6.2. Configuring Instances of an Enterprise Application in a Spreadsheet DSL

6.2.1. Overall Objectives of the Configuration DSL

- Give a bird's eye, editable and comparative view of the data points that determine the configurations.
- Include, in the same *workbook*, templates which describe common configuration artifacts (e.g. *properties* files).
- Make it simple to extend to other artifacts (XML, JSON, YAML amongst others).

6.2.2. Requirements for Configuration

When a service-oriented enterprise application is developed, there is usually a requirement to install the application in a number of different instances for developer and QA testing, for pre-production, for performance testing and for production installations. These installations will have somewhat different configurations.

Configurations can become quite thorny, especially with the proliferation of micro-service applications. Configuration Management has become an enterprise in and of itself. [10]

In the DSL for Configuration Management, I had a number of objectives:

- Have a central inventory of all instances, test and production
- Keep all the changeable values within a single Workbook, or, even better, on a single Worksheet
- Facilitate the comparison of the parameters in the different instances
- Have a separate Worksheet for each `properties` file that we generate with a template containing the properties that are generated
- Facilitate the generation of other types of types of formats that could be required (ex: *JSON*, *YAML* and *XML).*

A *Java* properties file has the following general look. Properties are a map from a string key to a string value. For properties, the model will need to determine what property keys to include for a given instance and what value to assign to that key.

```
system.location=AUSTIN
jms.QUEUE_MGR=DGBLHFCMP1
jms.HOST_NAME=gbltstfiag.yoyodyne
jms.PORT=23400
. . .
```

The *wrapper* properties file has a similar look but some additional requirements. All property keys are prefixed with `wrapper.` and the properties that are numbered require unique, consecutive numbering, hence the property generator need to manage this numbering.

```
wrapper.java.additional.1=-Drmi.hostname=localhost
wrapper.java.additional.2=-Xms1024m
wrapper.java.additional.3=-Xmx1024m
wrapper.app.parameter.1=classpath:yoyodyne_service.xml
. . .
```

6.2.3. A Model Specifying The Configuration of Instances of a System

The Workbook model for this DSL involves a main *worksheet* (i.e. **Instance** *worksheet*) which contains all of the structuring parameters of a system. This is presented as a series of tables (separated by blank lines) with the name of each table being at the upper-left corner of the table. The ***Instance*** table is required and contains the list of instances. Only these instances can be generated by this mechanism. The tables in this example (e.g. **QueueBroker**, **QueueSet**, etc) that follow the **Instance** table are **Linked** tables to which an instances refer. The name of a **Linked** table corresponds to the name of a column in the **Instance** table. The graphical connectors in the Spreadsheet are added to illustrate the linkages between the **Instances** table and the **Linked** tables.

There are multiple reasons for the **Linked** tables:

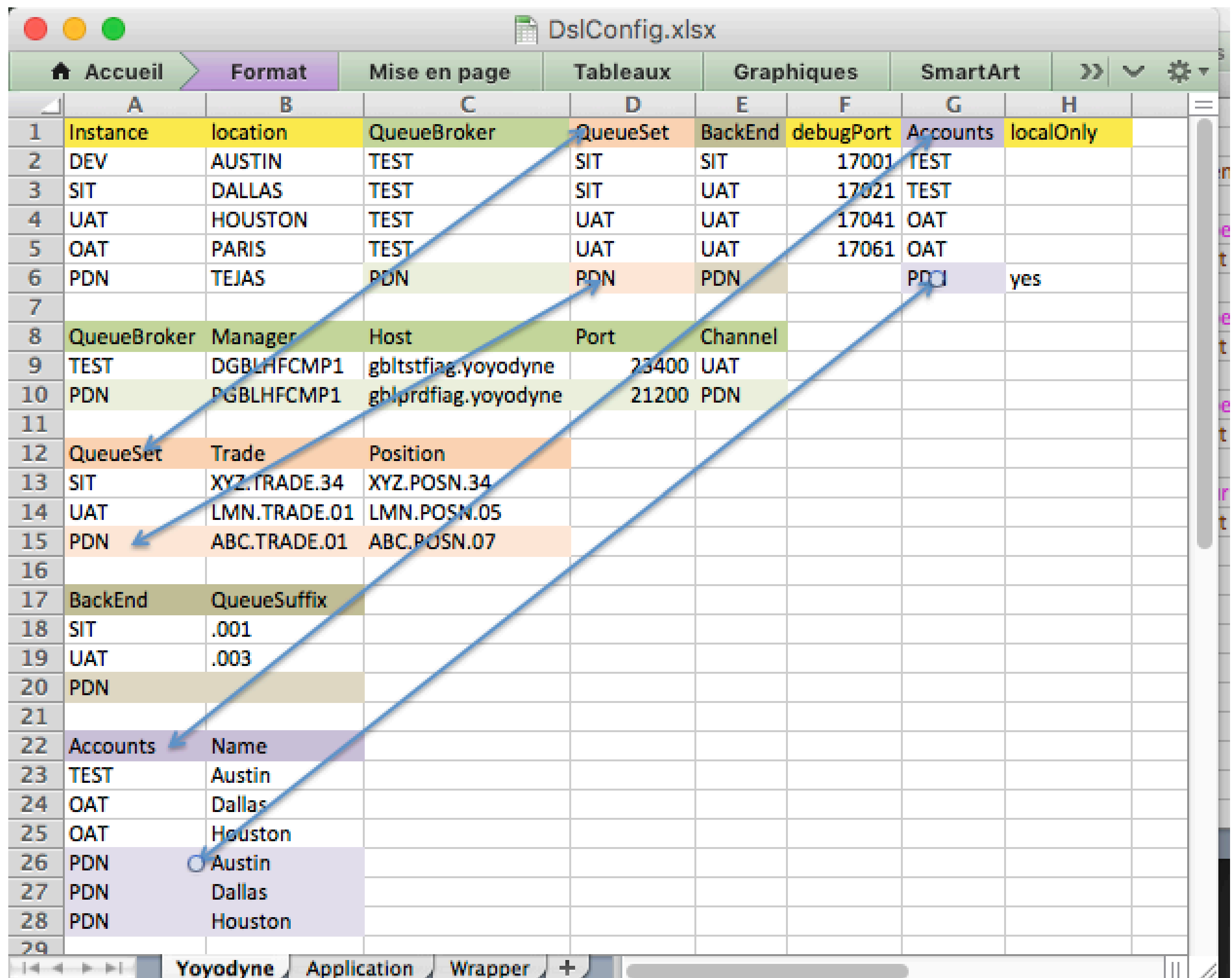
- They allow for regrouping information that belong together. (In the example, the four different parameters that correspond to the **QueueBroker** are in the same table).

- They give a symbolic name for a bit of data. (In the example, we associated the **QueueSuffix** *.003* with the **UAT BackEnd**. This documents the reason for the value *.003*).

- Allows the same data in the **Linked** tabled to be referenced multiple times from the **Instance** data (i.e. **DRY**[4]).

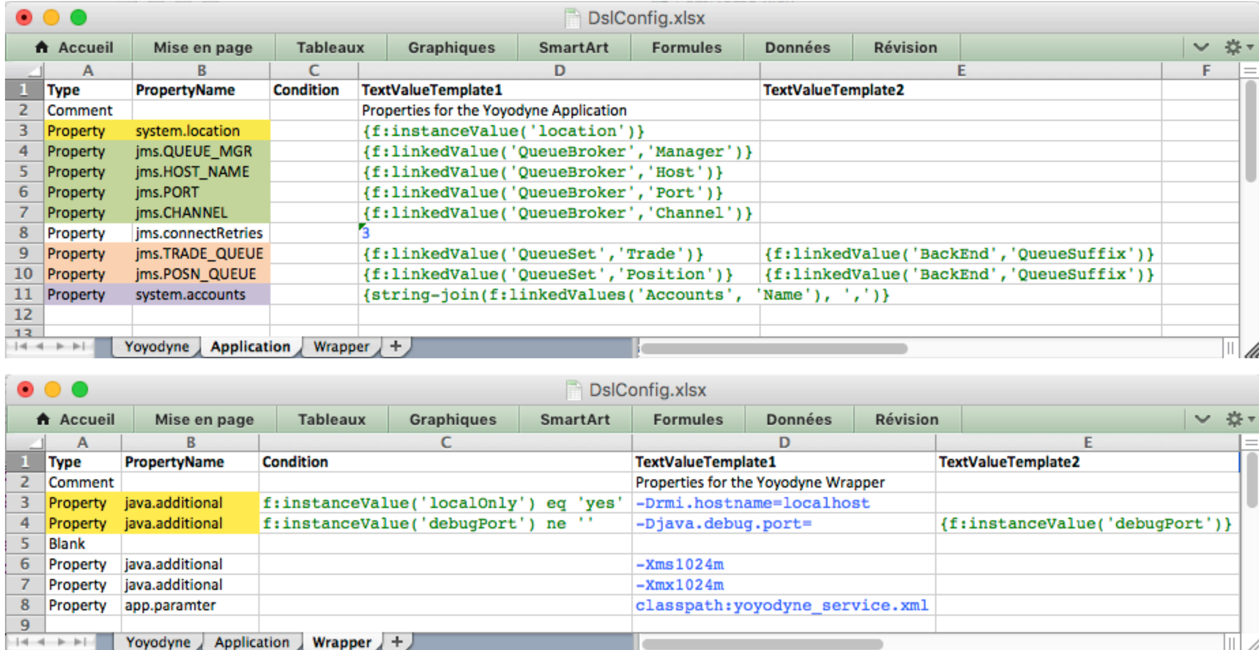
- Allows for one-to-many relationships from the **Instance** to the **Linked** data. (In the example, there are three **Accounts** for **PDN**.)

[4]https://en.wikipedia.org/wiki/Don%27t_repeat_yourself



The second and third worksheets correspond to two different `properties` files that need to be generated for an instance's installation. These worksheets are tables of types of *lines* to be produced in the resultant property files and the *rules* for generating them. The ***Condition*** column is a *boolean xpath* expression which determines whether or not this line should be produced in the results (where *blank* implies *true*).

The TextValueTemplate entries were manually formatted to help distinguish between *constant* values (in a shade of blue in the image) and the values calculated from the first page (in a shade of green in the image). The **PropertyName** cells have background colors that correspond to the separate regions of the first worksheet in order to facilitate their relation visually. The shade of yellow background color corresponds to information from the *Instance* columns as opposed to the *Linked* columns.



	A	B	C	D	E	F
1	Type	PropertyName	Condition	TextValueTemplate1	TextValueTemplate2	
2	Comment			Properties for the Yoyodyne Application		
3	Property	system.location		{f:instanceValue('location')}		
4	Property	jms.QUEUE_MGR		{f:linkedValue('QueueBroker','Manager')}		
5	Property	jms.HOST_NAME		{f:linkedValue('QueueBroker','Host')}		
6	Property	jms.PORT		{f:linkedValue('QueueBroker','Port')}		
7	Property	jms.CHANNEL		{f:linkedValue('QueueBroker','Channel')}		
8	Property	jms.connectRetries		3		
9	Property	jms.TRADE_QUEUE		{f:linkedValue('QueueSet','Trade')}	{f:linkedValue('BackEnd','QueueSuffix')}	
10	Property	jms.POSN_QUEUE		{f:linkedValue('QueueSet','Position')}	{f:linkedValue('BackEnd','QueueSuffix')}	
11	Property	system.accounts		{string-join(f:linkedValues('Accounts', 'Name'), ',')}		

	A	B	C	D	E
1	Type	PropertyName	Condition	TextValueTemplate1	TextValueTemplate2
2	Comment			Properties for the Yoyodyne Wrapper	
3	Property	java.additional	f:instanceValue('localOnly') eq 'yes'	-Drmi.hostname=localhost	
4	Property	java.additional	f:instanceValue('debugPort') ne ''	-Djava.debug.port=	{f:instanceValue('debugPort')}
5	Blank				
6	Property	java.additional		-Xms1024m	
7	Property	java.additional		-Xmx1024m	
8	Property	app.paramter		classpath:yoyodyne_service.xml	

The functions available for *xpath* expressions in the properties worksheets can come from the DSL library or can be user-supplied:

```
f:instanceValue(...)
f:linkedValue(...)
f:linkedValues(...)
```

6.2.4. Generating the Configuration Artifacts

In addition to the *workbook* containing the **Instance** and **Properties** *worksheets*, the configuration author will provide an *XSLT* which acts as the main entry point into the generation process. This stylesheet receives standard arguments for the generation process: the *URL* of the *workbook*, the name of the **Instance** *worksheet*, the *URL* of the root output directory for writing the generated files and the name of the **Instance** to be generated.

For the above example generating two properties files, the following *XSLT* stylesheet is sufficient. This stylesheet includes the common functions from an external XSLT file `DryGen.xslt`. It calls two supplied functions, one for generating standard Properties and another for writing Wrapper Properties, this latter property file requiring special treatment by adding number suffixes to the property keys in order to render them unique.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="3.0">

    <xsl:include href="DryGen.xslt" />
```

```
    <xsl:param name="workbookURL"     as="xs:string" required="yes" />
    <xsl:param name="worksheetName"   as="xs:string" required="yes" />
    <xsl:param name="instanceName"    as="xs:string" required="yes" />
    <xsl:param name="destinationURL"  as="xs:string" required="yes" />

    <xsl:template name="dslconfig" >
        <xsl:sequence select="f:writeProperties($workbookURL,
$worksheetName,
                                                $instanceName,
'Application',
                                                $destinationURL,
'yoyodyne.properties')" />

        <xsl:sequence select="f:writeWrapper   ($workbookURL,
$worksheetName,
                                                $instanceName, 'Wrapper',
                                                $destinationURL,
'yoyodyne.conf',
                                                $numberProperties)" />
    </xsl:template>

    <xsl:variable name="numberedProperties"
as="element(numberedProperty)*" >
        <numberedProperty name="java.classpath"  />
        <numberedProperty name="java.additional" />
        <numberedProperty name="app.parameter"   />
    </xsl:variable>

</xsl:stylesheet>
```

The implementation of `DryGen.xslt` itself is not provided here but can be described generally as a library which can read the different worksheets from the workbook and also implements the common functions used for generating properties and extracting instance and linked values. `DryGen.xslt` needs to extract *xpath* expressions from the worksheet and then evaluate the results of those expressions. There are a few different ways of going about this in the implementation:

- Use the `xsl:evaluate` element[5] in *XSLT3* (or the `saxon:evaluate()` function)[6]
- Generate an *XSLT* stylesheet containing the extracted *xpath* expressions and execute that stylesheet to obtain the results.

[5] https://www.w3.org/TR/xslt-30/#dynamic-xpath
[6] https://www.saxonica.com/html/documentation/extensions/functions/saxon-extension-functions.html

Not all versions of the *XSLT* implementations provide for dynamic *xpath* evaluation, sometimes for licensing reasons, hence the second option may be required in an implementation.

6.2.5. Generating non-Properties Configuration Artifacts

Because the main entry point to the generation process is the *XSLT*, it's easy to add extra functionality within that stylesheet in order to generate additional configuration artifacts, espection *XML*, *JSON* and *YAML* artifacts that are so in vogue these days.

6.2.6. Added Benefits of the Configuration DSL

Because the model requires only two files (i.e. the Worksheet and the Entry Point XSLT file) it's easier to deploy than N configuration files. Moreover, the configuration generation can be implemented *Just-In-Time*, often as part of automated deployment scripts.

Because this mechanism generates the configuration, we can add standardized unit tests that confirm the generated content..

6.3. Extracting Tabular Data from XML Documents

6.3.1. High-level Objectives

- Simplify the representation and authoring of business rules that determine how tabular data is to be extracted from structured data (XML) of diverse vocabularies and content models
- Facilitate the authoring by subject matter experts
- Simplify the mechanism for determining which business rules apply to which incoming document.
- Simplify the mechanism for determining the number of result lines to produce for an incoming document.

6.3.2. The Requirements for Extracting Tabular Data from XML

One problem that I ran across was a situation where a large number of diverse, detailed XML content models representing different financial instrument types and originating systems needed to be processed in quasi-realtime and have a fixed set of columns but a variable number of rows of tabular data extracted from each document. The number of rows extracted could be different for each document as a function of the type and nature of the instrument. Each line would have the same number of columns of data, each column representing a particular bit of information such as: coupon-rate, maturity-date, currency, amounts, etc.

The requirements of the DSL in this case were:

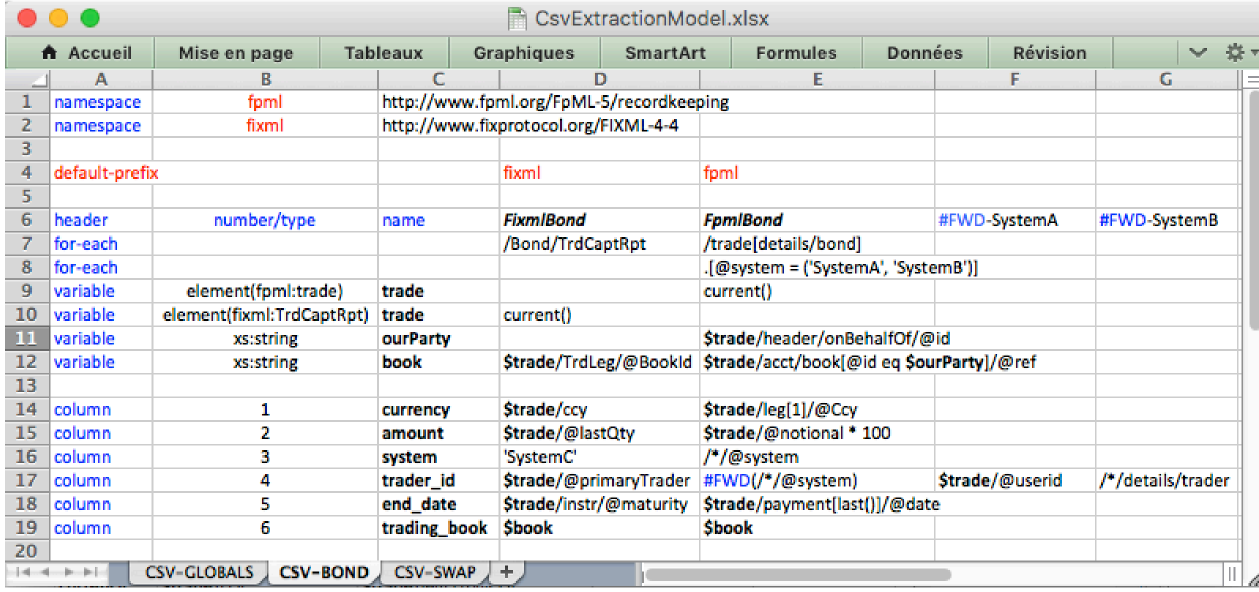
- Given that the Business Analysts would know the XML vocabularies for the received content
- Have a method of managing the extraction rules where Business Analysts could keep them updated and, moreover, test them for correctness
- The business rules would need to be able to determine, upon document reception, which rules to use for that particular document (ex: instrument type, vocabulary) and how many result lines would be produced.

6.3.3. The Model for Describing the Extraction of the XML Data

The basic mechanism for this model is the following:

- From the *workbook* an XSLT *stylesheet* is generated that contains a an ordered list of *named templates*.
- To extract data from a document, the document will be presented to the first named template in ordered list. If that template produces no output CSV lines, then the document will be presented to the next template in the list, and so on until there is a template that produces at least one line of output data.
- The line or lines of output data produced by the first template that did not produce 0 lines is the output data.

The order of the templates determined by the order of the *worksheets* in the *workbook* (left to right order) and by the *document types* declared within each *worksheet* in the *header* declaration. An example showing one *worksheet* with two *document types* (i.e. **FixmlBond** and **FpmlBond**, from the **header** line) is presented below.



CsvExtractionModel.xlsx

	A	B	C	D	E	F	G
1	namespace	fpml	http://www.fpml.org/FpML-5/recordkeeping				
2	namespace	fixml	http://www.fixprotocol.org/FIXML-4-4				
3							
4	default-prefix			fixml	fpml		
5							
6	header	number/type	name	FixmlBond	FpmlBond	#FWD-SystemA	#FWD-SystemB
7	for-each			/Bond/TrdCaptRpt	/trade[details/bond]		
8	for-each				.[@system = ('SystemA', 'SystemB')]		
9	variable	element(fpml:trade)	trade		current()		
10	variable	element(fixml:TrdCaptRpt)	trade	current()			
11	variable	xs:string	ourParty		$trade/header/onBehalfOf/@id		
12	variable	xs:string	book	$trade/TrdLeg/@BookId	$trade/acct/book[@id eq $ourParty]/@ref		
13							
14	column	1	currency	$trade/ccy	$trade/leg[1]/@Ccy		
15	column	2	amount	$trade/@lastQty	$trade/@notional * 100		
16	column	3	system	'SystemC'	/*/@system		
17	column	4	trader_id	$trade/@primaryTrader	#FWD(/*/@system)	$trade/@userid	/*/details/trader
18	column	5	end_date	$trade/instr/@maturity	$trade/payment[last()]/@date		
19	column	6	trading_book	$book	$book		
20							

CSV-GLOBALS | CSV-BOND | CSV-SWAP

Figure 2. Csv Extraction Model Spreadsheet

Running the example *worksheet* will generate two *named templates* with the names that are a composition of the *worksheet* name and the *document type* name from the `header` declaration. The generated templates follow:

```
<xsl:template xmlns:fpml="http://www.fpml.org/FpML-5/recordkeeping"
              xmlns:fixml="http://www.fixprotocol.org/FIXML-4-4"
              xpath-default-namespace="http://www.fixprotocol.org/
FIXML-4-4"
              name="f:BOND-FixmlBond" as="xs:string*">
    <xsl:for-each select="/Bond/TrdCaptRpt">
        <xsl:variable name="trade" as="element(fixml:TrdCaptRpt)"
select="current()" />
        <xsl:variable name="book" as="xs:string" select="$trade/TrdLeg/
@BookId" />
        <xsl:variable name="resultCells" as="item()*">
            <xsl:sequence select="f:empty-if-absent($trade/ccy)" />
            <xsl:sequence select="f:empty-if-absent($trade/@lastQty)" />
            <xsl:sequence select="f:empty-if-absent('SystemC')" />
            <xsl:sequence select="f:empty-if-absent($trade/
@primaryTrader)" />
            <xsl:sequence select="f:empty-if-absent($trade/instr/
@maturity)" />
            <xsl:sequence select="f:empty-if-absent($book)" />
        </xsl:variable>
        <xsl:value-of separator="{$separator}" select="for $i in
$resultCells
                                                    return f:encode-
csv($i, $separator)" />
    </xsl:for-each>
</xsl:template>



<xsl:template xmlns:fpml="http://www.fpml.org/FpML-5/recordkeeping"
              xmlns:fixml="http://www.fixprotocol.org/FIXML-4-4"
              xpath-default-namespace="http://www.fpml.org/FpML-5/
recordkeeping"
              name="f:BOND-FpmlBond" as="xs:string*">
    <xsl:for-each select="/trade[details/bond]">
        <xsl:for-each select=".[@system = ('SystemA', 'SystemB')]" >
            <xsl:variable name="trade" as="element(fpml:trade)"
select="current()" />
            <xsl:variable name="ourParty" as="xs:string" select="$trade/
header/onBehalfOf/@id" />
            <xsl:variable name="book" as="xs:string" select="$trade/acct/
book[@id eq $ourParty]/@ref" />
            <xsl:variable name="resultCells" as="item()*">
```

```
                <xsl:sequence select="f:empty-if-absent($trade/leg[1]/
@Ccy)" />
                <xsl:sequence select="f:empty-if-absent($trade/@notional
* 100)" />
                <xsl:sequence select="f:empty-if-absent(/*/@system)" />
                <xsl:choose>
                    <xsl:when test="/*/@system eq 'SystemA'" >
                        <xsl:sequence select="f:empty-if-absent($trade/
@userid)" />
                    </xsl:when>
                    <xsl:when test="/*/@system eq 'SystemB'" >
                        <xsl:sequence select="f:empty-if-absent(/*/
details/trader)" />
                    </xsl:when>
                </xsl:choose>
                <xsl:sequence select="f:empty-if-absent($trade/
payment[last()]/@date)" />
                <xsl:sequence select="f:empty-if-absent($book)" />
            </xsl:variable>
            <xsl:value-of separator="{$separator}" select="for $i in
$resultCells
                                                                    return
f:encode-csv($i, $separator)" />
        </xsl:for-each>
    </xsl:for-each>
</xsl:template>
```

The association between the generated templates and the example worksheet is illustrated in the following image:

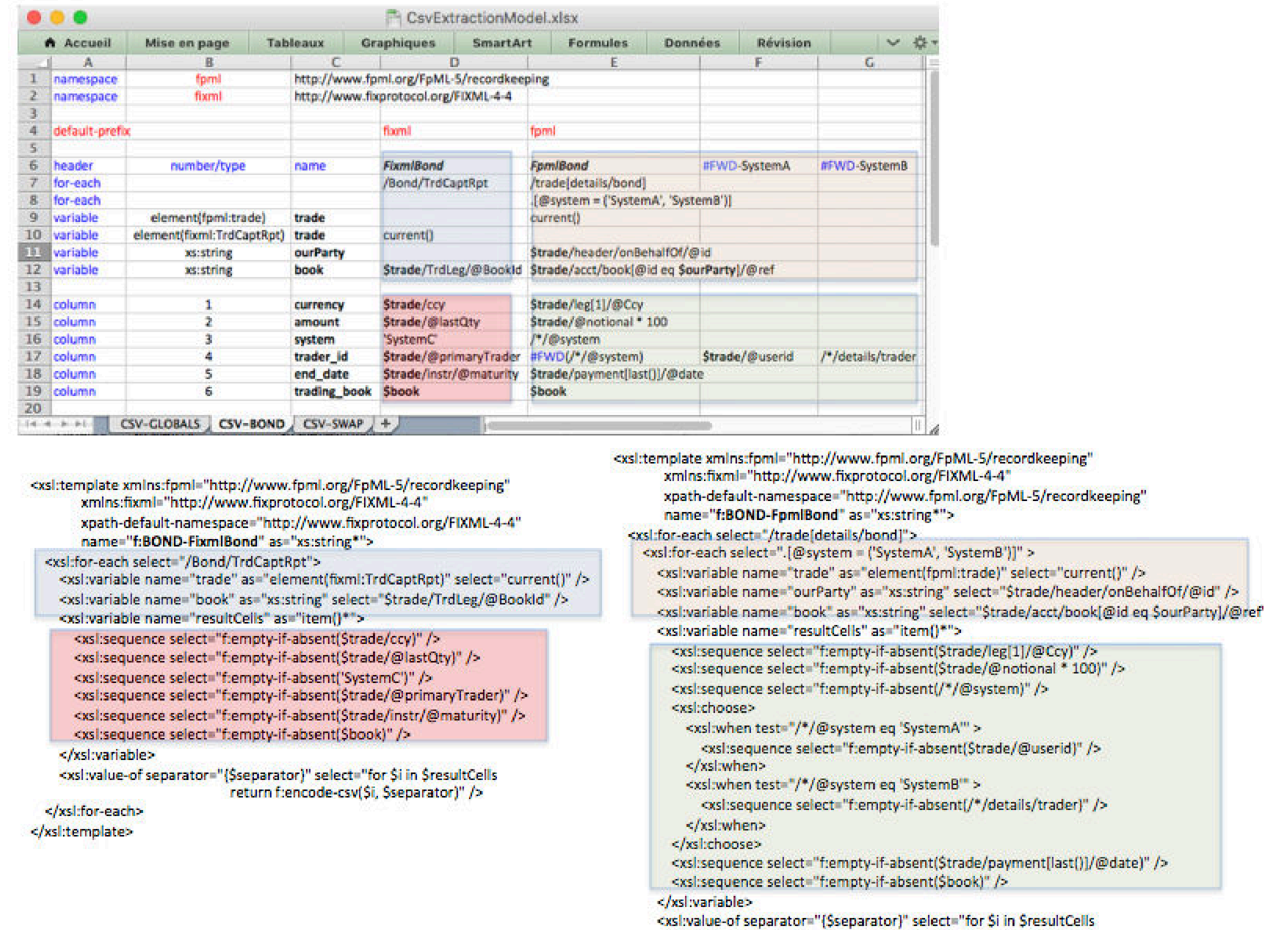


Figure 3. Association between the generated templates and the example worksheet

Within each *worksheet* with a name starting with **`CSV-`** there are a series of line declarations of different types, with the type given by the value in `Cell[1]` of each line:

- `header` — gives the names of the document types, used to form the names of the generated templates, from `Cell[4]` and greater (skipping columns for which the `Cell` value is empty or starts with a `'#FWD-'` value).
- `default-prefix` — gives the prefix associated with the namespace `URI` that will be set as the **xpath-default-namespace**
- `for-each` — each `Cell[]` in the same column as one of the `document types` from the `header` line, if it is not empty, is the value that will be used in a `select` attribute of a `xsl:for-each` element
- `variable` — declares an `xsl:variable` with an optional **type** declaration (`Cell[2]`) and with a **name** (`Cell[3]`)

- `column` — declares the column **number** (`Cell[2]`), the column **name** (`Cell[3]`) and the xpath expression (`Cell[position() ge 4]`) that is evaluated to to provide the string result for that column
- `#FWD(..)` — if the column rule appears to be a call to the function `#FWD(...)` then this really a **Forward** operation that chooses one of the rules in the following columns that are titled #FWD-XXX on the `header` line. The substring following the #FWD header column must match the string value of the xpath expression in the `#FWD(...)` pseudo-function call.

6.3.4. Testing and Analyzing the Data Extraction

The steps involved in authoring the DSL in a workbook and checking the output is illustrated in the following diagram. Here the Business Analyst will update the workbook and save it. To check the results, the Business Analyst can launch a batch process that runs two successive XSLT processes:

- A first XSLT process that reads the workbook and generates the result XSLT. This result XSLT is the *artifact* that will ultimately be used in production if it passes all tests.
- A second XSLT process that reads a stylesheet that is a *test harness* whose utility is to indicate the list of test documents to be transformed into the test output. The test harness will *xsl:include* or *xsl:import* the Generated XSLT from the first step.

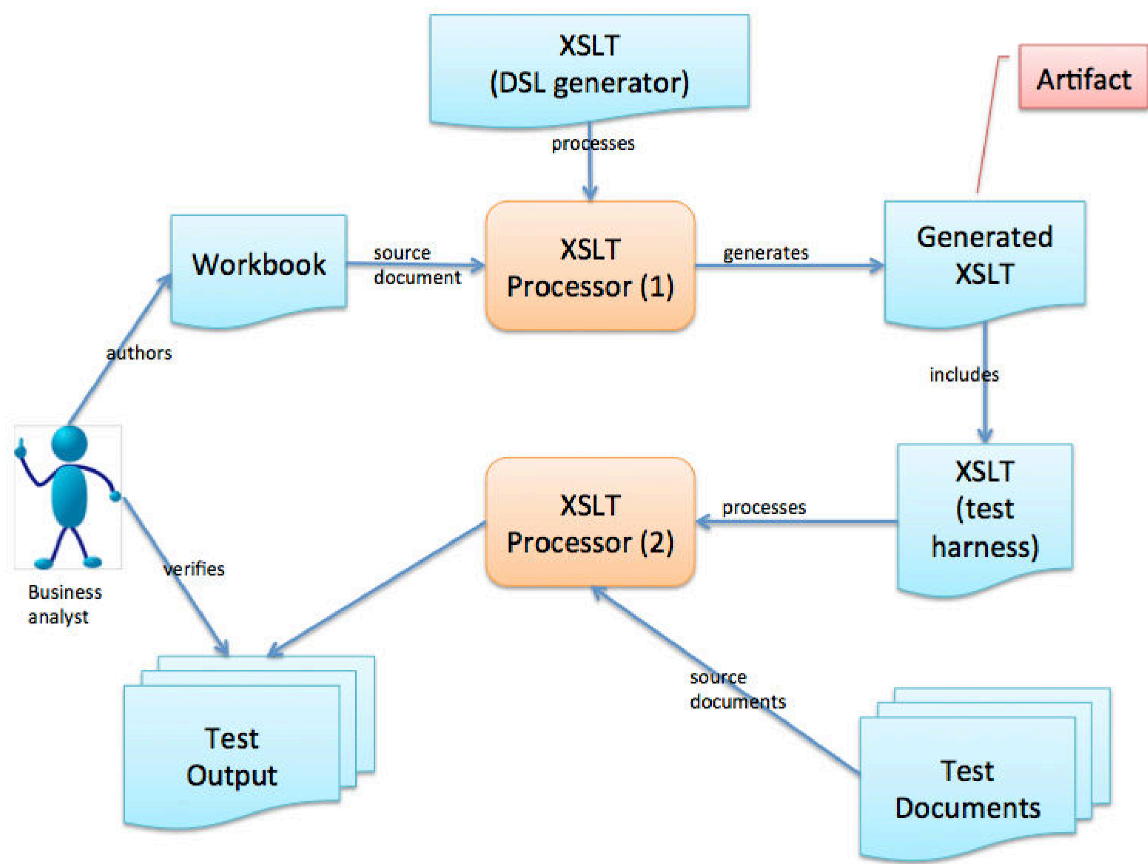


Figure 4. Illustration of the CSV Extraction Authoring and Testing Process

What is important about this process is that the Business Analyst has all the necessary tools for generating and verifying the output directly, without requiring additional technical help. This makes a Business Analyst a first-class contributor to the development process.

Note that in the first step, the XSLT process is generating an XSLT stylesheet as output. The generated XSLT is a separable deliverable that can be tested separately.

6.3.5. Business Expert Usage of the CSV Extraction DSL

This Spreadsheet DSL has seen good usage by a handful of business analysts and subject matter experts on practically a daily basis over the past 6 years as of this writing. One very nice result of this organization is that the rules for all the different types of instruments that we receive are presented in a *Rosetta Stone* fashion. In one workbook, our biggest, we have 85 columns of extracted data with 15 worksheets and 23 different generated templates. There are a number of other workbooks for other uses. Business Analysts can typically pick up the structure of the spreadsheet fairly quickly. The XPATH rules seem to be fairly intuitive for

Business Analysts with some analysts being aware of thornier XML issues such as document ordering.

Not only is this tool useful for data extraction but also for interactive data discovery. It's very easy to add new columns with new rules for doing things like adding sums of values, checking hypotheses, hence the tool becomes an interactive workbench.

Overall, the worksheet DSL imparts a structure to the extraction process that a generalized language such as XSLT would probably not have given.

6.4. Schema to Schema Translations

6.4.1. Overall Objectives

- Define the process of transforming from one XML vocabulary (XML schema) to another for a same domain.
- Provide a support for authoring and validation by subject matter experts.

6.4.2. Requirements

I was working on a system for which different types of credit facilities (i.e. loans) were modeled by a Front Office system and also by a Risk system. Both systems used XML Schema for describing their models, which were quite complex, managing a large number of different notions such as:

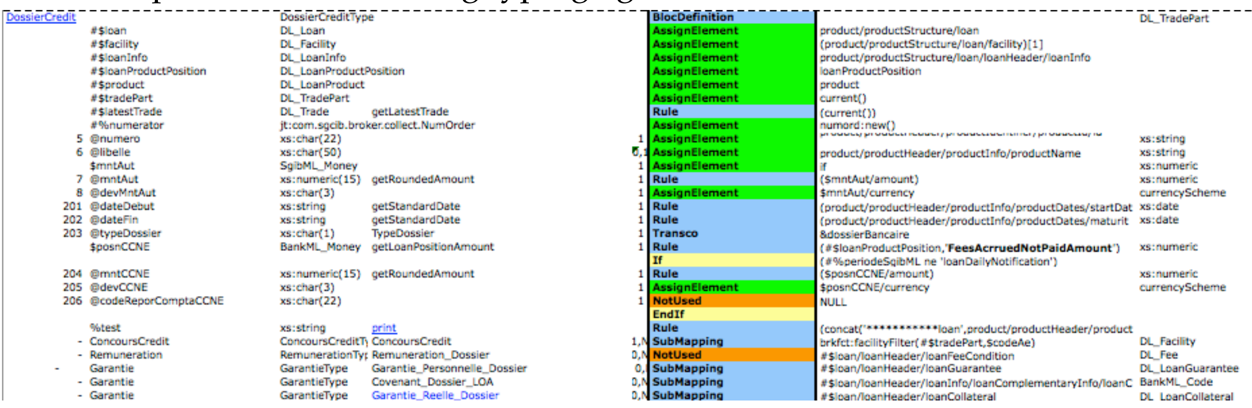
- Types of loans (fixed-rate, indexed, revolving credit, etc)
- Types of counterparties
- Types of collateral (buildings, airplanes, commodities, etc)
- Types of guarantees (cash, securities, insurance, export/import banks, etc)

Not only were there two different content models, but each model had a different set of Subject Matter Experts, hence getting the SMEs to agree was the major requirement. What we needed was a way of describing the transformation from the source system to the target system in business terms that could be understood, authored and validated by the SMEs themselves. The result was the DSL in a Workbook describing the transformation which then generated an XSLT.

In the biggest blocks, we were producing templates in schema-aware XSLT 2.0 that produced an element of a specific type. All parameters were strongly typed and all templates had required types hence the content model was closely monitored by the Schema Aware processor. This gave immediate feedback to the Business Analyst whenever there was either an xpath expression that did not conform to the input model or an xslt element that did not conform to the result model. An example:

```
    <!-- ================================= -->
    <!-- ContreGarantie_Concours: (99) -->
    <!-- ================================= -->
    <xsl:template match="element(*,bankml:DL_Reference)" as="element(*,
fsc2:GarantieType)"
                  mode="ContreGarantie_Concours" >
        <xsl:param name="elementName" as="xs:string" required="yes" />
        <xsl:param name="facility" as="element(*,bankml:DL_Facility)*"
tunnel="yes" />
        <xsl:param name="loanInfo" as="element(*,bankml:DL_LoanInfo)*"
tunnel="yes" />
        <xsl:element name="{$elementName}" type="fsc2:GarantieType" >
            <xsl:variable name="contreGarantieExterne" select="'13013'"
as="xs:string*" />
            <xsl:variable name="contreGarantieConsolide"
select="'13018'" as="xs:string*" />
            <xsl:variable name="isBranchOffice" as="xs:boolean"
                          select="((brkfct:getParty(current()))/
tradePartyType eq 'BranchOffice')"  />
```

The example shows that strong typing against the schema definition is used.



	DossierCredit	DossierCreditType			BlocDefinition		DL_TradePart
	#$loan	DL_Loan			AssignElement	product/productStructure/loan	
	#$facility	DL_Facility			AssignElement	(product/productStructure/loan/facility)[1]	
	#$loanInfo	DL_LoanInfo			AssignElement	product/productStructure/loan/loanHeader/loanInfo	
	#$loanProductPosition	DL_LoanProductPosition			AssignElement	loanProductPosition	
	#$product	DL_LoanProduct			AssignElement	product	
	#$tradePart	DL_TradePart			AssignElement	current()	
	#$latestTrade	DL_Trade	getLatestTrade		Rule	(current())	
	#%numerator	jt:com.sgcib.broker.collect.NumOrder			AssignElement	numord:new()	
5	@numero	xs:char(22)		1	AssignElement	[illegible]	xs:string
6	@libelle	xs:char(50)		0,1	AssignElement	product/productHeader/productInfo/productName	xs:string
	$mntAut	SgibML_Money		1	AssignElement	if	xs:numeric
7	@mntAut	xs:numeric(15)	getRoundedAmount	1	Rule	($mntAut/amount)	xs:numeric
8	@devMntAut	xs:char(3)		1	AssignElement	$mntAut/currency	currencyScheme
201	@dateDebut	xs:string	getStandardDate	1	Rule	(product/productHeader/productInfo/productDates/startDat	xs:date
202	@dateFin	xs:string	getStandardDate	1	Rule	(product/productHeader/productInfo/productDates/maturit	xs:date
203	@typeDossier	xs:char(1)	TypeDossier	1	Transco	&dossierBancaire	
	$posnCCNE	BankML_Money	getLoanPositionAmount	1	Rule	(#$loanProductPosition,'**FeesAcrruedNotPaidAmount**')	xs:numeric
					If	(#%periodeSgibML ne 'loanDailyNotification')	
204	@mntCCNE	xs:numeric(15)	getRoundedAmount	1	Rule	($posnCCNE/amount)	xs:numeric
205	@devCCNE	xs:char(3)		1	AssignElement	$posnCCNE/currency	currencyScheme
206	@codeReporComptaCCNE	xs:char(22)		1	NotUsed	NULL	
					EndIf		
	%test	xs:string	print		Rule	(concat('***********loan',product/productHeader/product	
-	ConcoursCredit	ConcoursCreditTy	ConcoursCredit	1,N	SubMapping	brkfct:facilityFilter(#$tradePart,$codeAe)	DL_Facility
-	Remuneration	RemunerationTyp	Remuneration_Dossier	0,N	NotUsed	#$loan/loanHeader/loanFeeCondition	DL_Fee
-	Garantie	GarantieType	Garantie_Personnelle_Dossier	0,	SubMapping	#$loan/loanHeader/loanGuarantee	DL_LoanGuarantee
-	Garantie	GarantieType	Covenant_Dossier_LOA	0,N	SubMapping	#$loan/loanHeader/loanInfo/loanComplementaryInfo/loanC	BankML_Code
-	Garantie	GarantieType	Garantie_Reelle_Dossier	0,N	SubMapping	#$loan/loanHeader/loanCollateral	DL_LoanCollateral

The result is quite detailed and looks a lot like a template in an XSLT. On the left hand side we have the attributes, elements and variables that we are producing from the information that is presented on the right-hand side. A SubMapping is really a call to `xsl:apply-templates`. A "Rule" is a call to a `xsl:function`. To give more information about the definitions:

- `#$XXX` — produce a variable that will be then be passed as a tunnel variable to all subsequent `xsl:apply-templates` from that template
- `$XXX` — produce a strongly-typed variable
- `@XXX` — produce an attribute
- `SubMapping` — `xsl:apply-templates` on a particular `select` value, passing all *variables* decl

- `Rule` — simply a call to a named function.

There is a separate page in the spreadsheet for declaring XPATH functions.

getFacilityNewSyndicat	_facility _partyRef _tradePart _shareType	element(*, defiml:DL_Facility) element(*,defiml:DL_Reference) element(*, defiml:DL_TradePart) xs:string	(($_tradePart/trade)[1]/specificTradeConditions/loanSpecificTradeConditions/loanT
getDistinctDLRefs	_dlRefs	element(*,defiml:DL_Reference)*	for $href in distinct-values($_dlRefs/@href) return (($_dlRefs[@href = $href])[1])
getPartRef	_tradePart _facility _dlRefs	element(*, defiml:DL_TradePart) element(*, defiml:DL_Facility) element(*,defiml:DL_Reference)*	for $d in $_dlRefs return (brkfct:riskPartGreater($_tradePart, $_facility,$d))

Also what was important was the translations from the code lists used in the source documents to the target schema.

COMMENT TranscoName	fromFieldName	fromCode	fromCodingS	toFieldName	toCode
SeniorityType-To-senioriteCreance	SeniorityType	JuniorSubordinated		senioriteCreance	JSO
SeniorityType-To-senioriteCreance	SeniorityType	Mezzanine		senioriteCreance	SSO
SeniorityType-To-senioriteCreance	SeniorityType	Senior		senioriteCreance	SEN
SeniorityType-To-senioriteCreance	SeniorityType	SeniorSecured		senioriteCreance	SEN
SeniorityType-To-senioriteCreance	SeniorityType	SeniorUnsecured		senioriteCreance	SEN
SeniorityType-To-senioriteCreance	SeniorityType	Subordinated		senioriteCreance	SSO
SeniorityType-To-senioriteCreance	SeniorityType	SuperPriority		senioriteCreance	SUP
SeniorityType-To-senioriteCreance	SeniorityType	Unknown		senioriteCreance	SEN
COMMENT	Transco SeniorityType-To-senioriteCreance				

6.4.3. Benefits of Generating the XSLT from the Schema-to-Schema DSL

The major benefit in this case was the possibility of have a fairly readable version of an XSLT within a Spreadsheet such that Subject Matter Experts could verify and validate the rules. One additional column in each line of the DSL was reserved for setting "Validated" on a rule-by-rule basis. This allowed the SMEs to follow the progress of Validation in a finely-grained fashion.

A major benefit was from the use of Schema Aware processing and strongly-typed template declarations. On the one hand, this tied the XSLT production directly to the two vocabularies. Furthermore, the SME authors benefitted directly and immediately from the Schema Aware checking of the XSLT Processor.

7. Conclusions

DSLs can be useful for application development and it's fairly straightforward to develop them in spreadsheets with XML Technology. I've presented some examples of usages of DSLs that I've developed, including two that involved close association and engagement of Business Experts. The result was the streamlining of development and a better participation of SMEs in the development chain.

Whether or not a DSL is appropriate or useful for a given problem remains a *Domain-Specific* determination. Furthermore, whereas the spreadsheet format may not be the optimal format for a given DSL, I've presented a few for which spreadsheet format was probably ideal. I suspect that the utility of spreadhsheet formats is not rare.

I have run into cases where SMEs were reluctant to get involved in a process that appeared to them to be more one of *technical development*; however, this has been, for me, the exception rather than the rule. For the most part, I've seen SMEs enthusiastic about the possibility of symbolic manipulation and direct feedback.

8. Caveats

Spreadsheet documents require some special treatment within Version Control Systems, including *git*. Typically, it's not straightforward to merge a spreadsheet document with divergent changes, hence manual merging will be required. Furthermore, visualizing the historical changes performed upon a spreadsheet document will not be within the capabilities of a text-based differences visualizer.

Bibliography

[1] Arie van Deursen, Paul Klint, and Joost Visser. *Domain-Specific Languages: An Annotated Bibliography*. 2000. CWI, The Netherlands. http://www.st.ewi.tudelft.nl/arie/papers/dslbib.pdf.

[2] S. I. Feldman. *Make --- A Program for Maintaining Computer Programs*. 1979. Bell Laboratories.

[3] S.C. Johnson and R. Sethi. *YACC: A parser generator*. 1990. Unix Research System Programmer's Manual, Tenth Edition, Volume 2.

[4] B.W. Kernighan. *A Typesetter-Independent TROFF*. 1982. Bell Labs.

[5] Tim Berners-Lee. *Hypertext Markup Language - 2.0*. November 1995. MIT/W3C. https://tools.ietf.org/html/rfc1866.

[6] Juha-Pekka Tolvanen and Steven Kelly. *Effort Used to Create Domain-Specific Modeling Languages*. ACM/IEEE. ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18). October 2018. MetaCase. http://www.metacase.com/papers/effort-create-domain-cameraReady.pdf.

[7] Marjan Mernik, Jan Heering, and Anthony M. Sloane. *When and How to Develop DomainSpecific Languages*. ACM/IEEE. ACM Computing Surveys, 37, 4. 2015. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.654&rep=rep1&type=pdf.

[8] Martin Fowler. *Domain-Specific Languages*. Pearson Education. Addison-Wesley Signaure Series. 2010.

[9] Dr. Michael Kay. *XSLT 2.0 and XPath 2.0, 4th Edition*. Wrox. 2008.

[10] Susan Dart. Carnegie-Mellon University. *Concepts in configuration management systems.*. ACM. ftp://ftp.sei.cmu.edu/pub/case-env/config_mgt/papers/

cm_concepts.pdf. Proceedings of the 3rd international workshop on Software configuration management. 1991.

[11] John Ellson, Emden Gansner, Eleftherios Koutsofios, Stephen North, and Gordon Woodhull. AT&T Labs. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*. Springer-Verlag. https://graphviz.gitlab.io/_pages/Documentation/EGKNW03.pdf. 2004.

How to configure an editor

An overview of how we built Fonto

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

Abstract

In 2012 a web agency took on the challenge of building an XML editor. This paper gives an overview of a number of concepts that proved to be useful, and some concepts that did not.

1. Introduction

FontoXML is an editor for XML document or structured content in more general terms. It's primary intended to be used by Subject Matter Expert who do not necessarily have any knowledge of XML. FontoXML editor is much a platform then a shrink-wrapped piece of software. It can be tailored to support any kind of XML document format, including DITA 1.3, JATS , TEI, Office Open XML and other proprietary format. The platform itself is highly confirurable so it can be tailored to specific use cases.

All of these different editors have three different parts of configuration:

- the schema defines which elements may occur where;
- elements are assigned to 'families' causing them to receive a visualization and basic cursor behaviour;
- and operations, which define the effect of the toolbar buttons and hotkeys.

Especially the operation part is where FontoXML has seen some major API redesigns. This paper will discuss a number of key decisions we've made and where the XML manipulation layer of FontoXML will move to. We hope that this paper will give an insight in how a small team of JavaScript developers with medium knowledge of XML technologies made the platform on which a large number of XML editors have been and are being built.

2. Iteration 0

FontoXML started in late 2012 when a publisher of commentary on legislative content moved to use an XML schema to encode their content in. Their authors were happily working in MS Word and threatened the company with leaving for their competitor if the new tooling would hinder them in any way. The publish-

ing company required a solution where the user experience is the number one priority and where anything technical would just be taken care of. At that moment, there were no existing XML editors which met their requirements. This is the birth of the precursor of FontoXML, which started life as a specific, bespoke solution.

Writing a user interface containing a toolbar, designing a way to render XML as an HTML view, updating the view as efficiently as possible and many other parts of this version of FontoXML were fun and challenging to do, but this paper will focus on the XML manipulation side of the editor: what happens when I press a button?

2.1. The beginning: canInsert

The first iteration of the XML manipulation was as simple as it gets. There were **commands** which could be queried for their state: whether they were enabled or disabled or whether they were 'active'. Getting state was implemented separately from actually executing the command. Schema validation was hard-coded and exposed functions that could check whether adding an element could exist at a given location. The command then had to implement an execute method to perform the intended mutation if it is allowed. At this point, every command had to query the system by itself. There were no atomic transactions: when a command inevitably had to try multiple things, the command had to undo all of them when it found out that it could not run to completion.

2.2. Schemata as regular expressions

One of the major hurdles of writing an XML editor in JavaScript is the absence of most XML tools for the browser environment. When we started to work on the editor in 2012, there was no usable XPath engine, no usable DOM implementation (besides the browser implementations, which all had a number of inconsistencies, browser-specific issues and performance issues). An XML-schema implementation was also not available. However, XML Schema is a **regular tree grammar**???, meaning the content models of an element are regular languages. We used actual JavaScript regular expressions as a basic schema validator:

```
// The note element may contain p, ol or ul elements, repeated
indefinitely
const noteSchema = /((<p>)|(<ol>)|(<ul>))*/;
// To allow us to work with regular expressions,
// we need to 'stringify' the contents of an element.
function createElementContentString (element) {
  return element.children.map(child => '<' + child.nodeName +
'>').join('');
}
```

```
const noteElement = document.createElement('note');
noteElement.appendChild(document.createElement('p'));
const isValid = noteSchema.test(createElementContentString(nodeElement));
// isValid is true

noteElement.appendChild(document.createElement('note'));
const isValid = noteSchema.test(createElementContentString(nodeElement));
// isValid is false, a note may not contain a note
```

This string-based validation allows us to compose a *'canInsert'* function quite elegantly:

```
function canInsert (element, offset, nodeName, schemaRegex) {
  const elementContents = element.children
    .map(child => '<' + child.nodeName + '>');
  // Use the Array#splice method to add a new element at the given offset
  elementContents.splice(offset, 0, '<' + nodeName + '>');
  return schemaRegex.test(elementContents.join(''));
}
```

We used the browser's built-in RegEx engine to evaluate these expressions.

Even though this approach is a very pragmatic and easy way to implement a schema validator, it is not the best way. The regular expressions were hand-written and hard to maintain. Besides the maintainability issue, string regular expressions are not very extensible. 'Repairing' a split element by generating missing elements turned out to be one of the most code-intensive parts of the editor and the regular-expression approach to validation could not give us any information that we could use as input for this problem.

Perhaps surprisingly, performance was not a bottleneck when we used regexes for schema validation. One would expect the string allocations would cause performance problems when validating elements with a large number of children. We did not run into these issues because the documents we loaded were relatively small. Nodes with relatively many children still only had dozens of child elements, not hundreds.

3. Iteration 1

3.1. Validation

As the first iteration of the XML schema validator, we opted to still use a regular language-based implementation, but to implement our own engine for running them[7]. By compiling the schema to a non-deterministic finite automaton (NFA), we can check whether our element is valid[2].

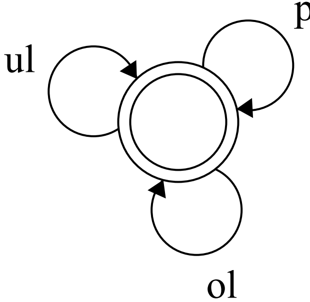


Figure 1. Example validation state machine

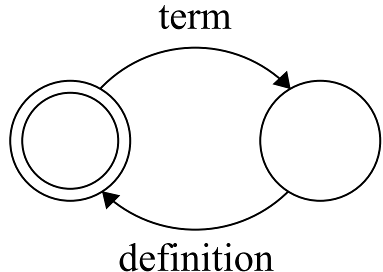


Figure 2. Example of a state machine for a repetition schema

3.2. Synthesis

The schema for which we initially developed the editor was relatively simple and was specialized for different departments of our client. A figure could contain a title followed by an image and a list had to contain list items. During the development of the editor, the schema grew into different sub-schemas. Some of them forced or disallowed titles in figures, while some added new elements which better expressed their specific content. This required us to make the commands we had better interpret the schema, to see which choices could be made where.

We needed to not only know whether an element is valid or not, but we also needed to know WHY it is deemed invalid and whether we can ‘fix’ it by creating required elements at the correct positions. A major advantage of the new NFA-based schema validator is extensibility. By tracking the path(s) taken though the NFA that lead to its acceptance of a given input sequence, we can see which decisions led to the acceptance or rejection. This can, in turn, tell us which elements are missing under a node.

This process essentially adds an additional state transition type: *record*, which does not 'eat' an input but leaves data on the trace:

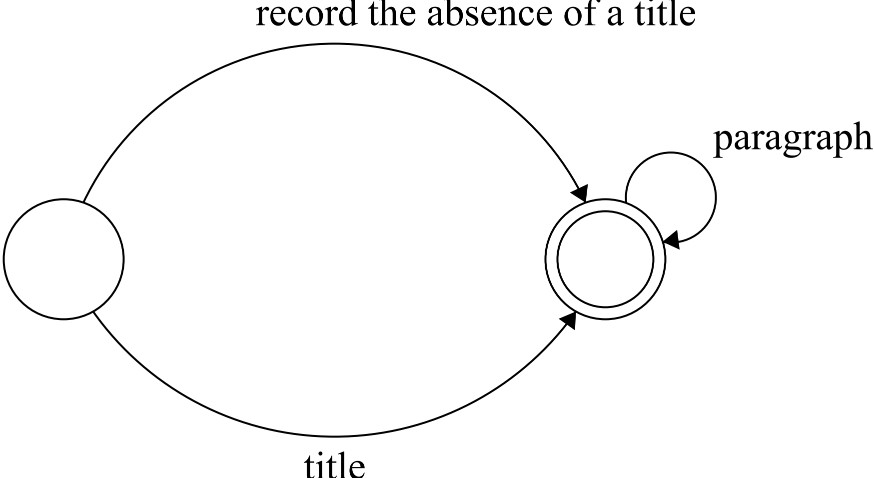


Figure 3. Diagram of the state machine describing the content model (title paragraph*)

When we validate an element containing a single `<paragraph>` with the content model `(title paragraph*)`, it will not be valid. If we however use synthesis, the *record* branch will be taken. This will leave us with the information that we are missing a `<title>` element, and the exact position at which to insert it in order to make the document valid again.

We call this process synthesis and even though randomly creating elements may result in an unexpected situation, this happens relatively rarely in practice, so we still depend on it.

3.3. Flow behaviour

The second hurdle when writing more abstract XML mutations is defining an abstract flow behaviour. Authors expect that having a cursor inside an image should be prevented and that pressing the right arrow with the cursor before a footnote shoulder should not cause the cursor to appear inside the footnote. However, the way images and footnotes are represented as XML varies from schema to schema.

We call the set of properties of elements that define such global navigation and editing behaviour flow properties. Among other things, these directly determine the effect of pressing enter or backspace, as well as the cursor behaviour in and around these elements. The most important flow properties of an element are:

- **Splittability**: Whether the element should be cut in two then we press enter in it, or when we insert a new element in it which can not be contained. A para-

graph is usually splittable, while a link or a semantic element like a TEI `<persName/>` is not. This property also defines whether we can merge an element with similar ones, to keep the merging and splitting behaviour consistent.

- **Closed**: Whether we can move in and out of this element using the cursor keys. Footnotes are closed and do not influence the cursor behaviour around them.
- **Detached**: Similar to **closed**, but with additional concern that it is fully detached from its parent and should act like it isn't there. Elements containing metadata are detached.
- **Removable if empty**: Whether the element should be deleted if it is empty and backspace is pressed in front of it. `<note>` elements in DITA can be *removable-if-empty*: if one presses backspace when the cursor is in an empty note, it should be removed.
- **Auto-mergeable / auto-removable if empty**: Empty HTML `<b>` elements are useless, and two adjacent `<b>` elements are equivalent to a single element spanning their combined content. FontoXML normalizes them when it finds them being empty or adjacent.
- **Default text container**: If any, what element should be created when one starts to type in it? For example, typing inside an empty <note> may need to first create a <paragraph> to contain the new text

3.4. Blueprints

Asking for validity or trying to come up with the definitive structure before changing anything in the dom did not allow for very manageable code, so we came up with a new component: blueprints.

A blueprint is basically an overlay over the DOM, it intercepts all accesses to the DOM (like retrieving the parent of a node, or which attributes it has). It also intercepts the mutations to the underlying DOM, so that they don't affect the underlying "real" DOM, but the updated relations in the blueprint can still be queried.

```
const blueprint = new Blueprint();
const newElement = document.createElement('newElement');
blueprint.appendChild(document.documentElement, newElement);
const newParent = blueprint.getParentNode(newElement);
// newParent is now set the documentElement node.

blueprint.setAttribute(newElement, 'attr', 'value');
const attrValue = blueprint.getAttribute(newElement, 'attr');
// attrValue is now set to the string 'value'
```

```
const isValid = validator.isValidNode(blueprint, newParent);
// The validator is passed the blueprint so it can see
// whether the new structure is valid

if (isValid) {
  // We know it's valid, so we can safely apply these changes to the DOM
  blueprint.realize();
}
```

This approach allows us to make atomic changes to the DOM; we can revert everything by just not applying the new relations. The programming model for these commands is more maintainable and easier. Instead of trying to predict what a command will result in, the command can just apply changes and query them.

Because these changes are atomic, we can implement a very simple ‘getState’ function for all of these commands. Instead of seeing whether a command can work, we can just run the normal command, validate the outcome and simply not apply its changes to the actual DOM.

Bueprints let us make a proposed change to the dom. We extend this concept to ‘overlays’, which are essentially blueprints over blueprints. By using these overlays, we can compose different ‘atomic’ changes to essentially ‘puzzle’ our way through a mutation; making edits as we go along, but being able to revert them as we go along.

3.5. Schema-independent primitives: vertical insert nodes

Blueprints and overlays let us make changes to the document and revert them. This caused the birth of schema-independent primitives like ‘insertNode’, which attempts to insert a node by splitting the ancestry:

```
Vertical insert nodes:

Let $position be the position at which we will insert,
  consisting of a container node and an offset
Let $node be the new node we'd like to insert

A:
Start an overlay on the blueprint to contain all the changes in this
mutation.

B:
If $position resides in a text node:
Split the text node at the given offset
Set $position to after the first half of the result
```

```
C:
Start an overlay on the blueprint
Insert the $node at the given position
Validate the parent of $node, using the new relations present in the
blueprint

If the result is valid:
  Apply all pending overlays on the blueprint
  Return true to indicate success

D:
If the result is invalid:
  Discard the overlay on the blueprint to revert the insertion of $node
  If the container of the $position is the document element:
    Discard the blueprint overlay and return false to indicate failure.
  Otherwise:
    Split the container of the $position at the given offset
    Insert the new second half of the container to after the first half.
    Set the $position to after the first half and before the second half
      of the old container.
    Continue at section C.
```

3.6. Operations

At this point in time, these mutations were composed using JavaScript. This works well for most mutations, but soon we found ourselves re-inventing the same commands over and over and that flowing data from things like modals to command were cumbersome to implement. We currently still use a component that we invented during that timeframe: operations.

Operations are pipelines that allow composing XML manipulating “commands”, data-modifying “transforms”, modals and other operations. Arguments for the command, which is usually the last step, flow through the pipeline and are modified and appended to by each of the steps.

As with commands, operations support a “getState” mode, which performs a dry-run in order to determine the validity of an operation without affecting the current state of the application.

3.6.1. JSONML

These operations are written in JSON files, which pose another challenge. XML does not let itself be inlined all too well in a format like JSON. Line breaks in JSON strings are disallowed and it’s easy to forget to escape a double quote somewhere. We often use JSONML[4]when an XML fragment should be made serializable or could be included in JSON at some point, like the operations pipeline.

JSONML also blends in nicely with our existing data interpolation functionality in operation.json files:

```
{
  "insert-a-note": {
    "label": "Insert note",
    "summary": "insert a note, with the type attribute set from a modal",
    "steps": [
The ask-for-type modal will write to the 'type' of the
 data object we pass along in the pipeline.
      { "type": "modal/ask-for-type" },
      {
        "type": "command/insert-node",
        "data": {
          "childNodeStructure": [
            "note",
            {
The following line lets us set the 'type' attribute to
 the value of the modal
              "type": "{{type}}"
            }
          ]
        }
      }
    ]
  }
}
```

The main limitation of the operation pipeline lies in predictability; an operation does not explicitly define which keys of the 'data' object it reads from, nor does it define to which keys it will write. We attempted to resolve this using documentation, but in practice, this does not adequately address the problem.

4. Iteration 2

4.1. Families, Content Visualization Kit

Editors using the FontoXML platform were growing to cover more and more different schemata, but all elements were configured all of the different 'flow' properties separately. Different FontoXML implementations 'felt' differently and were getting inconsistent in their behaviour of similar elements. However, most elements represented the same 'concept' within a document. Some act like blocks of text, some act like unsplittable, semantic inlines. We call these roles 'families', which each define a set of flow properties and a visualization, intended to make its properties predictable for an author.

These families are made up of three different groups:

- **Containers**,which contain blocks
- **Blocks**, which contain lines
- **Inlines**, which can contain other inlines and text

These different groups consist of different families, which may have their own sub-families:

- **Frame**: An unsplittable entity, which contains blocks. They are ignored for navigation next to the default text container so that the arrow-keys directly jump between the blocks it contains. Frames are visualized with a border to indicate they can't be split using the enter key.
- **Sheetframe**: A sub-family of **frames** are sheet-frames, which represent whole documents or units of work. They are the root element in FontoXML, the white 'sheets' in which one types.
- **Block**: A 'block' of text, like an HTML `<p/>` element. They are splittable, not detached, removable if they're empty, but not automatically removable, nor automatically mergeable. They are not ignored for navigation, but their parents' elements often are.
- **Inline formatting**: Like the HTML `<b/>`, `<i/>`, `<sup/>` elements. They may be split, do not influence the behaviour of the enter keys but are automatically mergeable.
- **Inline frames**: The inline variant of a **frame**. Also unsplittable, but they can directly contain text.

4.2. Selectors

Nodes are assigned a family using a system inspired by CSS selectors, using a JavaScript fluent API:

```
// Configure elements matching the
// 'self::span[@type="bold"] and parent::span' XPath selector
configureAsInlineFormatting(
  matchNodeName('span')
    .withAttribute('type', 'bold')
    .requireChild(matchNodeName('span')),
  'Parent of nested span')
```

If an element matched multiple configurations, we chose one based on the 'specificity[5]' of the selector. The specificity algorithm is based on CSS selector specificity.

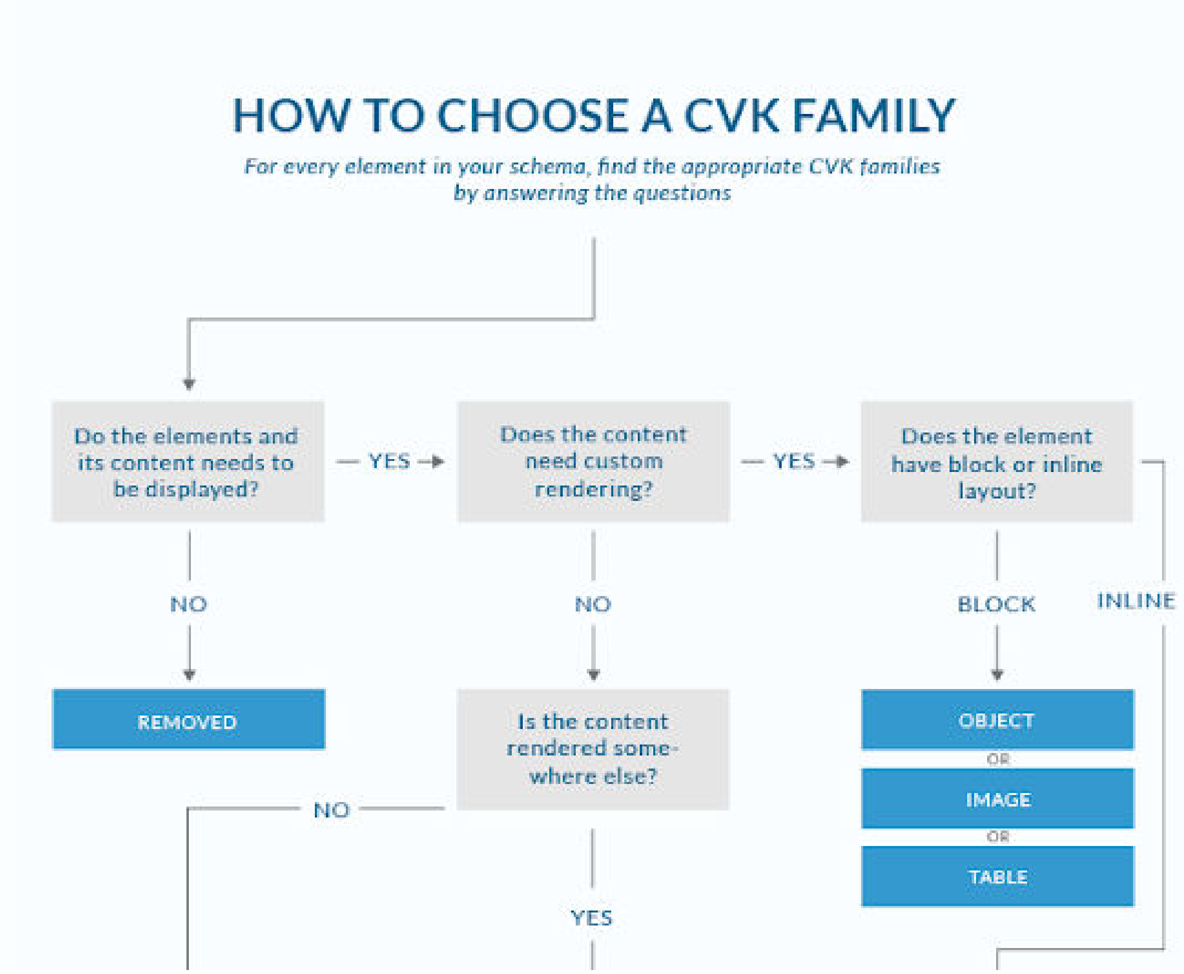


Figure 4. Excerpt of the "CVK family chart"

4.3. Stencils

The new generic commands were powerful when combined with the operations pipeline, but they were hard to target. Provisioning modals with data also started to be very code-heavy. We needed a new component that could read from and write data to the dom: stencils.

Stencils are an extension on JSONML, with regex-like ‘gaps’ that can catch nodes. They can be used much like XPath, but they describe a structure that can be matched much like a schema. For instance, a stencil matching a <figure> element, catching the contents of the <title> element and the value of the @href attribute looks like this:

```
[
  "figure",
  [
    "title",
    [{
      "bindTo": "titleGap",
```

```
      "multiple": true,
      "required": false
    }]
  ],
  ["img", { "href": { "bindTo": "hrefGap"}}]
]
```

This stencil would allow both reading from and setting the value of the title using the “titleGap” gap.

Stencils fulfil another role as well: matching a DOM structure. The following stencil matches a `<metadata>` element with a `<title>` (disregarding its contents) and a `<related-links>` element, which contains an empty `<link>` element with the `@href` attribute set to a certain value:

```
[
  "metadata",
  ["title", [{"multiple": true, "required": false}],
  [
    "related-links",
    {"multiple": true, "required": false},
    ["link", {"href": "a certain value"}],
    [{"multiple": true, "required": false}]
  ]
]
```

This stencil matches the same structures as the XPath expression `self::metadata[title and related-links/ link/ @href = "a certain value"]`: match all `metadata` elements with a `title`, with a `link` to `"a certain value"`. The stencil is subjectively easier to manage in some cases because it can perform two-way data binding as well test whether a node aligns with it. Obvious downsides of these stencils are related to the complexity of them. Stencils were later enhanced by:

- allowing nested XPath expressions;
- omitting nodeNames, to allow us to match a node regardless of its name;
- requiring the selection to either be fully contained, start before, or end after a given node;
- setting the selection after the command was executed;
- and many more features.

This did make stencils more powerful, and the selection mechanics allowed them to match structures which XPath expressions could not. This did not make them easier to write, and we are working on deprecating this API in favour of more XPath and XQuery usage.

4.4. Extender

'Fuzzy commands' became one of the most powerful features of the FontoXML editor platform. These are primitives like InsertNodes: *'insert a node at the current cursor position, splitting anything that has to be split to allow for this new element to be placed'*. A new related need arose: *insert a new element somewhere under a given element, at any offset.* This can be computed using a new extension upon schema validation and synthesis: extension.

The exact inner workings of the extension component is too complex to cover in this paper and should be covered more in-depth in a future one.

5. Iteration 3 (now)

5.1. XPath

The editor API was quickly growing to using too many proprietary 'standards', which were invented by our own. This makes it difficult to explain to an outsider how to use our platform for their own editor. Far too many configurations were reverting to using custom JavaScript callbacks to express their concerns, for example when assigning elements to a family. This sparked the need of an XPath engine, preferably one that could leverage a number of optimizations we implemented for our current Selectors implementation. We also wanted to use the new features that the XPath 3.1 standard defines, which was in its Candidate Recommendation phase at the time. At that time, there were no usable implementations, so we wrote an open source XPath 3.1 JavaScript engine[3].

5.2. XPath observers

A number of the editors require knowledge of which kind of elements are present in a document, and what information is related to them. This information is used in, for example, a Schematron panel. We provide a way to detect which dom-relations an XPath query traverses, and a way to be notified when these relations have been altered, so we can update our user interface as few times as possible. This process is further explained in our earlier paper[6].

5.3. XQuery & Update Facility

At this moment, we are in the process of writing an XQuery 3.1 + XQuery Update Facility implementation, so that we can port our DOM manipulation algorithms to those standards and expose them from an XQuery API.

We are still learning how we can express concepts that are not present in those standards, like the selection, DOM positions, the blueprint concept, and 'fuzzy'

commands. These concepts are what makes writing editor commands different from for instance writing database update queries.

Database update queries are usually fully aware of the restrictions a schema enforces. This contrasts with editor commands where a certain kind of 'just make it work' mentality is required to correctly interpret what an author wants to do.

6. Conclusion

When one writes a platform for an XML editor in JavaScript, they should be prepared to write a lot of the XML tools themselves (though this may have changed in the last seven years). Furthermore, while it does pay off to design your own configuration languages due to the flexibility of them and the ability to keep it simple, using more complete standards like XPath and XQuery will work better in the long run because those technologies will inevitably have a larger community around them.

Bibliography

[1] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. 2005. Taxonomy of XML schema languages using formal language theory. ACM Trans. Internet Technol. 5, 4 (November 2005), 660-704.

[2] Implementing regular expressions: `https://swtch.com/~rsc/regexp/`

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

[4] `http://www.jsonml.org`

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

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

[7] `https://github.com/bwrrp/whynot.js`

Discover the Power of SQF

Octavian Nadolu
Oxygen XML Editor
<octavian_nadolu@oxygenxml.com>

Nico Kutscherauer
<kutscherauer@schematron-quickfix.com>

Abstract

In the last few years, the Schematron QuickFix (SQF) language has started to be used in more and more projects from various domains (technical publishing, automotive, government, financial). The Schematron QuickFix language was presented for the first time three years ago at the XML Prague conference. It has now reached a point where we have a second draft specification available on GitHub[1] *and within the W3C "Quick-Fix Support for XML Community Group*[2]*.*

Based on the feedback that we have received from the users, the Schematron QuickFix language was updated and new improvements where added. For example, the multilingual support available in Schematron can now also be used for SQF messages, the quick fixes can now be generated dynamically, the `sqf:keep` element was replaced by the `sqf:copy-of` element, and the `sqf:stringReplace` regular expression interpretation can now be controlled by the `@flags` attribute.

The SQF language is very powerful, you can use it to make multiple changes in the current document, or to make changes in external documents. You also can use XSLT[3] *code in a quick fix to make complex processing or you can display dialog boxes to get input from the user. You can create generic fixes, or you can define abstract quick fixes that can be implemented with different parameters depending on the context.*

Schematron quick fixes can be created for any type of XML document. There are quick fixes created for DITA, DocBook, or TEI documents, and also quick fixes for XSLT that check the quality of the XSLT code and propose solutions to correct the problems.

In this presentation, you will discover the new additions for the SQF language and how you can create useful and interesting quick fixes for the Schematron rules defined in your project. It will include examples of quick

[1] http://schematron-quickfix.github.io/sqf
[2] https://www.w3.org/community/quickfix/
[3] https://www.w3.org/TR/xslt/all/

fixes for several types of projects using abstract quick fixes to enable easier creation of specific fixes, or XSLT code for more complex fixes. You will also discover some use cases for quick fixes that extend your Schematron scope.

1. Introduction

Schematron is a powerful language, and one of the reasons is because it allow the schema developers to define there own custom messages. The messages can also includes hints to help the user correct the problem, but this operation must be performed manually. Correcting the problem manually is inefficient and can result in additional problems.

The SQF language allows you to define actions that will automatically correct the problem reported by Schematron assertions. This will save you time and money, and will help to avoid the potential for generating other problems.

2. Schematron QuickFix

Schematron QuickFix (SQF) it is a simple language that allows the Schematron developer to define actions that will correct the problems reported by Schematron rules. SQF was created as an extension of the Schematron language. It was developed within the W3C "Quick-Fix Support for XML Community Group". The first draft of the Schematron Quick Fix specification was published in April 2015, second draft in March 2018, and it is now available on the W3C Quick-Fix Support for XML community group[4] page.

The actions defined in a Schematron QuickFix are called operations. There are four types of operations defined in the SQF language that can be performed on an XML document: *add*, *delete*, *replace*, and *string replace*. The operations must perform precise changes in the documents without affecting other parts of the XML document.

Example 1. A Schematron Quick Fix that adds a 'bone' element as child of the 'dog' element

```
<sch:rule context="dog">
    <sch:assert test="bone" sqf:fix="addBone">
        A dog should have a bone.</sch:assert>
    <sqf:fix id="addBone">
        <sqf:description>
            <sqf:title>Add a bone</sqf:title>
        </sqf:description>
        <sqf:add node-type="element" target="bone"/>
```

[4] https://www.w3.org/community/quickfix/

```
    </sqf:fix>
</sch:rule>
```

3. SQF Use Cases

3.1. Quality Assurance

It is important to have quality control over the XML documents in your project. You do this using Schematron schema in combination with other schemas (such as XSD, RNG, or DTD). Schematron solves the limitation that other types of schema have when validating XML documents because it allows the schema author to define the tests and control the messages that are presented to the user. The validation problems are more accessible to users and it ensures that they understand the problem.

However, correcting the validation problems in the XML documents can sometimes be difficult for a content writer. Most of the content writers are not XML experts, but they know the context of the document very well and they are experts in their domain. If a problem occurs when they write content in the document, they need to fix it correctly without adding new ones.

For example, perhaps you have a rule that checks if two consecutive lists are found and reports any occurrences as a problem. This can happen if the list was split by mistake, or maybe the writer forgot to add a paragraph or sentence before the second list. If the content writer encounters this type of problem and tries to merge the lists manually, it is possible for them to make mistakes and introduce other problems.

Example 2. A Schematron assertion that reports a problem if two consecutive lists are found

```
<sch:rule context="ul">
  <sch:report test="following-sibling::element()[1][name() = 'ul']">
    Two consecutive unordered lists. You can probably merge them into
one.
  </sch:report>
</sch:rule>
```

On the other hand, an XML expert knows the syntax very well and knows how to fix the problem, but they may not be familiar with the domain of the content writer. In this case, since the XML expert and content writer will need to work together to correct the problems, this will introduce extra costs and take more time.

Using SQF, you can provide in-place actions to help the content writer correct the problems by themselves without involving the XML expert, and without pro-

ducing new errors. This will solve the problem faster and companies will spend less money. The XML expert can focus on developing new rules and quick fixes for the content writer.

Example 3. A Schematron QuickFix that merges two adjacent lists into one

```
<sqf:fix id="mergeLists">
  <sqf:description>
    <sqf:title>Merge lists into one</sqf:title>
  </sqf:description>
  <sqf:add position="last-child" select="following-sibling::element()[1]/
node()"/>
  <sqf:delete match="following-sibling::element()[1]"/>
</sqf:fix>
```

3.2. Efficiency

A Schematron QuickFix can also be used to improve efficiency when adding content in XML documents. You can define actions that will automatically add a new XML structure in the document at a valid location, or actions that will convert an XML structure into another. These types of actions will help the content writer add content more easily and without making mistakes.

For example, suppose you want to make sure that all of your documents have keywords (index terms) defined in a prolog so that searches in the output will generate desired results. For this, you can create a Schematron rule that reports a problem if the *keywords* element is missing from the prolog.

Example 4. Schematron rule that verifies if the keywords element is present

```
<sch:rule context="topic">
  <sch:assert test="exists(prolog/metadata/keywords)" role="warn">
    No keywords are set for the current topic.
  </sch:assert>
</sch:rule>
```

Then you can create an action that will automatically add an entire structure of a *prolog* element (with *metadata*, *keywords*, and *indexterm* elements) before the *body* element, and the content writer just needs to specify the keyword value in a user entry dialog box.

Example 5. Schematron QuickFix that inserts an XML structure

```
<sqf:fix id="addKeywords">
  <sqf:description>
    <sqf:title>Add keywords for the current topic</sqf:title>
  </sqf:description>
```

```
  <sqf:user-entry name="keyword">
    <sqf:description><sqf:title>Keyword value:</sqf:title></
sqf:description>
  </sqf:user-entry>

  <sqf:add match="body" position="before">
    <prolog>
      <metadata>
        <keywords>
          <indexterm>
            <sch:value-of select="$keyword"/>
          </indexterm>
        </keywords>
      </metadata>
    </prolog>
  </sqf:add>
</sqf:fix>
```

You can also create more complex actions. For example, actions that will correct a table layout and uses complex XSLT processing, or actions that use data from other documents or from a database. This will allow the content writer to focus on the content of the document while the quick fixes will help them to easily insert XML structure or to fix various issues that can appear during editing.

4. Abstract Quick Fixes

The Schematron QuickFix language is a simple language, and has just four types of operations that can be performed (add, delete, replace, and string replace). Being a simple language, it is easy to learn and use, and also easy to implement by applications.

Sometimes the developers that creates the quick fixes need to use other types of operations (such as wrap, unwrap, rename, or join). They expect to have these operations defined in the language. Defining more operations in the language will help them create the quick fixes more easy, but this means that the language will be more complicated to learn and harder to be implemented by applications. A solution to this problem is to define a library of generic quick fixes that can be used for other types of operations.

A library of quick fixes can be implemented using abstract quick fixes. An abstract quick fix can be defined as a quick fix that has abstract parameters defined at the beginning of an `sqf:fix` element.

Example 6. Schematron abstract quick fix that can be used to rename a generic element

```
<sqf:fix id="renameElement" role="replace">
    <sqf:param name="element" abstract="true"/>
```

```
    <sqf:param name="newName" abstract="true"/>
    <sqf:description>
        <sqf:title>Rename '$element' element in '$newName'</sqf:title>
    </sqf:description>
    <sqf:replace match="." target="$newName" node-type="element"
select="node()"/>
</sqf:fix>
```

An abstract quick fix can be instantiated from an abstract pattern. The pattern must have all the parameters declared in the quick fix, and the quick fix must declare all the abstract parameters that are used. Abstract parameters cannot be used as normal XPath variables. The reference of the abstract parameter will be replaced by the value specified in the abstract pattern.

Example 7. Schematron abstract pattern that reference an abstract quick fix

```
<sch:pattern id="elementNotAllowed" abstract="true">
    <sch:rule context="$element">
        <sch:assert test="false()" sqf:fix="renameElement">
            Element '$element' not allowed, use '$newName' instead.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

The abstract pattern can be instantiated by providing different values for the parameters. Therefore, you can quickly adapt to multiple variants of XML formats and provide rules and quick fixes that will allow the user to correct the problems.

Example 8. Schematron abstract pattern instantiation

```
<sch:pattern is-a="elementNotAllowed">
    <sch:param name="element" value="orderedlist"/>
    <sch:param name="newName" value="itemizedlist"/>
</sch:pattern>
```

Another solution for providing other quick fix actions without using abstract patterns is to use the `sqf:call-fix`[5] element.

5. Multilingual Support in SQF

The second draft of the Schematron QuickFix specification comes with an important addition, the localization concept for quick fixes. It is based on the Schematron localization concept, but it is more flexible.

[5] http://schematron-quickfix.github.io/sqf/publishing-snapshots/March2018Draft/spec/SQFSpec.html#param.call-fix

A new attribute was added for the `sqf:title` and `sqf:p` elements, the `@ref` attribute. In the value of the `@ref` attribute, you can specify one or more IDs that point to different translations of the current phrase. The specification does not restrict the implementations of the `@ref` attribute to a specific reference structure.

Example 9. Schematron QuickFix that has multilingual support

```
 <sqf:fix id="addBone">
     <sqf:description>
         <sqf:title ref="fix_en fix_de">Add a bone</sqf:title>
         <sqf:p ref="fix_d_en fix_d_de">Add a bone as child element</
sqf:p>
     </sqf:description>
     <sqf:add node-type="element" target="bone"/>
 </sqf:fix>
```

One possible implementation of the multilingual support in SQF is to use the Schematron diagnostics element. You can define a diagnostic for each referenced `id` and specify the language of the diagnostic message using the `xml:lang` attribute on the `sch:diagnostic` element or on its parent.

Example 10. Schematron diagnostics

```
 <sch:diagnostics>
     <sch:diagnostic id="fix_en" xml:lang="en">Add a bone</
sch:diagnostic>
     <sch:diagnostic id="fix_de" xml:lang="de">Fügen Sie einen Knochen
hinzu</sch:diagnostic>
 </sch:diagnostics>
```

This implementation conforms with the Schematron standard that also uses diagnostic for localization. It is easier to translate the messages because the Schematron messages and quick fix messages are kept together, and another important aspect is that it will be the same implementation used for both Schematron and SQF messages.

There are also some issues with this implementation. One of them is that you cannot have IDs with the same name in your document because the diagnostic uses XML IDs. Another issue is that SQF will depend on the Schematron language and cannot be encapsulated separately.

Another implementation of quick fix localization is to use Java Property Files. In this case, the localized text phrases should be stored in external files, grouped by language. These files should be placed parallel to the Schematron schema with the name pattern `${fileName}_${lang}.xml`. `${fileName}` should be the name of the Schematron schema but without the extension. The `@ref` attribute from the quick fix must reference the property key.

Example 11. Java Property File for German translation

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="dog.addBone.title">Füge einen Knochen hinzu</entry>
    <entry key="dog.addBone.p">Der Hund wird einen Knochen erhalten.</
entry>
</properties>
```

In contrast to the references to `sch:diagnostic` elements, in this case, it is not necessary to make any changes in the Schematron schema to introduce a new language. You just have to add a file with the name (for example `localized_fr.xml`) in the same folder of the Schematron file (for example `localized.sch`).

However, this needs a different implementation than the standard Schematron, and the Schematron messages and the SQF messages will be in different locations.

6. Generate Quick Fixes Dynamically

Another important addition in the second draft of the Schematron QuickFix specification is the possibility to define generic quick fixes. Using a generic quick fix, the developer can generate multiple similar fixes using the values provided by an XPath expression.

In the first draft of the SQF specification, it was not possible to have a dynamic amount of quick fixes for a Schematron error. The developer should specify the quick fixes presented for a Schematron error. He could only control whether or not a quick fix will be presented to the user by using the `@use-when` attribute.

To create a generic quick fix, the SQF developer needs to set the `use-for-each` attribute for the `sqf:fix` element. The value of the `use-for-each` attribute must be an XPath expression. For each item of the evaluated XPath expression, a quick fix is provided to the user. The context inside of the quick fix will be the current Schematron error context. To access the value of the current item from the XPath expression, a built-in variable `$sqf:current` can be used.

Example 12. A Generic QuickFix that provides a quick fix to remove each item from a list

```
<sqf:fix id="removeAnyItem" use-for-each="1 to count(li)">
    <sqf:description>
        <sqf:title>Remove item #<sch:value-of select="$sqf:current"/></
sqf:title>
    </sqf:description>
    <sqf:delete match="li[$sqf:current]"/>
</sqf:fix>
```

Using generic quick fixes, the SQF developer can now provide a dynamic number of fix action depending on a set of values from the current document or they can use an XPath expression to get the values from external documents.

7. Interactive Schematron through SQF

7.1. An ignore concept

One of Schematron's advantages is the ability to hierarchize and evaluate the rules you set up yourself. While structure-based schema languages such as XSD or RELAX NG only know valid or invalid documents, Schematron can differentiate between errors, warnings and information with the help of the role attribute – although the Schematron standard did not even provide this classification. It only became a de facto standard.

However, with these possibilities new questions and problems also arise. When does a document pass a Schematron validation? Can I ignore a Schematron assert that has been classified as a warning? Basically this should be possible, otherwise such a classification is quite pointless. An unfixed warning should not usually lead to a process abort.

At the same time, however, an unfixed warning is usually an indication that the document is not completely "clean". Warnings are often used to notify the user of problematic content that is often incorrect, but does not necessarily have to be false. This means that an unfixed warning often points to bad content, but not always.

The condition of a document containing warnings is similar to the condition of Schrödinger's cat: As long as I don't look at the cause of the warnings, the document is correct and incorrect at the same time. A manual intervention by the user is therefore required in order to check the warnings. Assuming that the user has fixed all problematic warnings, the state of the document is cleaned and all remaining warnings would be "ignorable". However, this cleaned state is extremely volatile because it is not saved anywhere. The persons involved must remember that the existing warnings can be ignored. However, this only applies until the document is processed again. Then it immediately falls back into the condition of Schrödinger's cat.

With a small number of ignorable warnings and editors of the document, this problem can still sometimes be handled. However, as soon as the number of ignorable warnings accumulates, this very quickly leads to all warnings being ignored – whether problematic or not. One option to mark warnings as ignorable is not to surrender to the convenience of the user, as you might think at first, but, if correctly applied, may improve the documents. In addition, an ignore concept can provide clarity about the status of a document by accepting documents with warnings that have not been ignored as incorrect.

7.1.1. How can an ignore concept work in Schematron?

The biggest challenge for an ignore concept in Schematron is that there are no uniform errors. Complex Schematron errors can occur due to interactions of different document structures in one or even in different documents. To mark an error as ignorable, it must be uniquely identified, otherwise also other errors could be mistakenly ignored.

Schematron errors also have in common that they all have a context (`sch:rule/@context`). This context is also usually used to specify the location of the error. If a help structure is added relative to the context, the Schematron can react to such a help structure and ignore a possible error. The context of a possible error is marked as "to ignore".

The help structure must be as closely connected as possible to the context. It should also be noted that, according to the Schematron standard, every node type can be a context, even if not every implementation supports it. A help structure could, for example, be an attribute in a special namespace or a Processing Instruction (PI) that is inserted before the context node.

Advantages of attributes are that a connection with an element as context is very strong and thus also easily catchable with Schematron. In addition, naming conflicts are excluded by a separate namespace. The disadvantages are that there is no possibility for PIs, comments or text nodes as context to uniquely assign attributes to it. A connection would be extremely loose. In addition, attributes in foreign namespaces can lead to validation errors in other schemata, which, for example, control the structure of the document.

PIs, on the other hand, can be set relative to any node type and should not conflict with any other schema, since they are ignored by default. On the other hand, the connection to the context node is rather loose – as soon as a node is moved, for example, it must be noted that the associated PI is also moved. The PI is also at a disadvantage when it comes to uniqueness, since it cannot have a namespace.

The value of the attribute or PI should contain an identifier of the error. A simple `d2t:ignore="true"` is not enough because ignoring an error does not mean that you want to ignore another error that happens to have the same context. It is recommended to use is a list of IDs, which refers to the IDs of the `sch:assert` or `sch:report` elements. If the patterns are not too large, the pattern ID is also sufficient.

7.1.2. XSLT style guide example

So much for dry theory. Now it shall be demonstrated how the concept works in practice. As an example serves an XSLT style guide which uses Schematron to check for certain spellings and constellations that are permitted by the XSLT

standard – i.e. they do not lead to compilation errors – but still do not correspond to a good language style.

Here, as an example rule, a template is presented that is not used anywhere else in the stylesheet:

```
<sch:pattern id="p_04_unused_items">
    <sch:rule context="xsl:template[@name]" role="warn">
        <!-- Check if this template is used somewhere in the stylesheet.
-->
```

Now XSLT offers the possibility to call a template initially. Then it would be correct if it is no longer used in the stylesheet. A warning could be ignored. However, since this is a very rare case, we do not want to do without this check.

In order to give the XSLT developer a way to ignore this check, we have now decided for a d2t:ignore help attribute in the namespace `http://data2type.de/`. The possible value should be the pattern ID of the error to be ignored. In our example this would look like this:

```
<xsl:template name="initial" d2t:ignore="p_04_unused_items"
xmlns:d2t="http://data2type.de/">
```

With a simple additional empty `sch:rule` you can catch the contexts to be ignored in the Schematron by using the order of the rules within a pattern:

```
<sch:pattern id="p_04_unused_items">
    <sch:rule context="*[@d2t:ignore = 'p_04_unused_items']"/>
    <sch:rule context="xsl:template[@name]" role="warn">
        <!-- Check if this template is used somewhere in the stylesheet.
-->
```

As a result, all elements having an `@d2t:ignore` attribute with the `p_04_unused_items` value are caught in this pattern by the first rule and ignored because the rule does not contain any tests. If you want to extend the logic so that an element can ignore different patterns, you should be allowed to specify a list of pattern IDs. In Schematron it would look like this:

```
<sch:rule context="*[tokenize(@d2t:ignore, '\s') =
'p_04_unused_items']"/>
```

7.1.3. Interaction with SQF

Now the above concept is very nice in theory, but unfortunately it has a big hook in practice: The user is required to have some prior knowledge to ignore such a Schematron warning. First of all, he has to know that such an ignore function exists at all. This may be quickly communicated to a single user. But if one imagines that such an XSLT style guide is used within a larger company, this basic information can quickly get lost deep in some software documentation. Even if the ignore function is known, it must still be known exactly how it works: it must

be an `ignore` attribute in the specific namespace `http:// data2type.de/` (not `http:// www.data2type.de/` or `http:// data2type.de`!). In addition, the corresponding pattern ID must be known for each Schematron error. In reality it leads to the fact that such a logic, if it is implemented at all, is quickly forgotten.

This is where Schematron QuickFix joins the game. All this information, which the user would otherwise have to remember, can be stored in a QuickFix by the Schematron developer. The Schematron error simply gets a QuickFix "Ignore this error". If the user selects such an "Ignore QuickFix", this fix makes exactly the changes that the user would otherwise have to make manually: it sets the `d2t:ignore` attribute in the correct namespace with the correct pattern ID.

Please note: If an Ignore QuickFix is embedded in an extensive QuickFix collection, it should no longer be a problem that it is not found. Furthermore, it will be easier for the user to use an Ignore QuickFix only if it is justified. A standalone Ignore QuickFix, on the other hand, gives the impression that ignoring a warning is the only possible solution.

In our style guide example, the Ignore QuickFix would look like this:

```
<sqf:fix id="ignore_p_04_unused_items">
    <sqf:description>
        <sqf:title>Ignore this error.</sqf:title>
    </sqf:description>
    <sqf:add match="." target="d2t:ignore" node-type="attribute"
             select="'p_04_unused_items'"/>
</sqf:fix>
```

A `d2t:ignore` attribute (`target="d2t:ignore" node-type="attribute"`) is added (`<sqf:add ... />`) to the current context node (`match="."`). The value used is the pattern ID `p_04_unused_items` (`select="'p_04_unused_items'"`).

In the version in which different patterns can be ignored, an existing `d2t:ignore` attribute should not be overwritten, but its value should be taken over:

```
<sqf:fix id="ignore_p_04_unused_items">
          <sqf:description>
              <sqf:title> Ignore this error.</sqf:title>
          </sqf:description>
          <sch:let name="newIgnoreId" value="string-join((@d2t:ignore,
'p_04_unused_items'), ' ')"/>
          <sqf:add match="." target="d2t:ignore" node-
type="attribute"              select="$newIgnoreId"/>
</sqf:fix>
```

7.2. XSD Guide

XML Schema (XSD) is a large, widely used standard for structuring XML data. As in many big standards, there are different ways to express the same in XSD. How-

ever, due to its widespread use, non-developers or developers with low XSD experience often have to deal with XML Schema. In addition, the XSD standard has a very verbose XML syntax in which even seemingly simple structures with a complex spelling must be described. This often leads to "Google-guided development", which usually results in a wild mishmash of XSD structures. If insufficient testing is added, schemas are created in which no one can say exactly what is allowed where and why.

The idea of the XSD Guide[6] is to provide the inexperienced user with a tool that will guide him to develop an XSD schema while adhering to the rules of a particular design pattern.

A design pattern in XSD restricts the extensive standard to certain structures. Well-known design patterns are Salami Slice, Venetian Blind or Russian Doll. While Salami Slice works with global element declarations that are referenced locally, Venetian Blind only declares the possible root elements globally and all others locally. Instead, Venetian Blind uses global types that are referenced by the local element declarations. Russian Doll, on the other hand, completely refrains from reuse and declares the document structure within a global element declaration.

7.2.1. Design patterns examples

Salami Slice:

```
    <xs:element name="document">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="head"/>
                <xs:element ref="body"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="head">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="title"/>
                <xs:element ref="meta" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="meta">
```

[6] https://github.com/octavianN/thePowerOfSQF/tree/master/Samples/XSDGuide

```
            <xs:complexType>
                <xs:attribute name="value" type="xs:string"/>
                <xs:attribute name="key">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:pattern value="\S+"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
        </xs:element>
        <xs:element name="body">

        </xs:element>
```

Venetian Blind:

```
    <xs:element name="document" type="documentType"/>
        <xs:complexType name="documentType">
            <xs:sequence>
                <xs:element name="head" type="headType"/>
                <xs:element name="body" type="bodyType"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="headType">
            <xs:sequence>
                <xs:element name="title" type="xs:string"/>
                <xs:element name="meta" minOccurs="0" maxOccurs="unbounded"
    type="metaType"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="metaType">
            <xs:attribute name="value" type="xs:string"/>
            <xs:attribute name="key" type="keyType"/>
        </xs:complexType>
        <xs:simpleType name="keyType">
            <xs:restriction base="xs:string">
                <xs:pattern value="\S+"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:complexType name="bodyType">

        </xs:complexType>
```

Russian Doll:

```
<xs:element name="document">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="head">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="title" type="xs:string"/>
                        <xs:element name="meta" minOccurs="0"
maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:attribute name="value"
type="xs:string"/>
                                <xs:attribute name="key">
                                    <xs:simpleType>
                                        <xs:restriction base="xs:string">
                                            <xs:pattern value="\S+"/>
                                        </xs:restriction>
                                    </xs:simpleType>
                                </xs:attribute>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="body">

            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

7.2.2. Using Schematron and SQF

The XSD Guide now has two tasks: The first is to check whether the XSD developer has violated the rules of the initially selected design pattern, and to notify him accordingly. On the other hand, simple and understandable functions for generating XSD structures are to be offered to the user. With these functions a kind of user guidance is to develop, which does not only let the user generate the necessary XSD structures in a simple way, but also makes sure that the selected design pattern is adhered to. In contrast to some code generators, not only functional code is to be generated, but also code that is maintainable and readable and can be further developed without the help of an XSD Guide.

As a language for very specific rules that can be queried with XPath, Schematron is ideal for the first task of the XSD Guide – checking the rules for the selected design pattern. Less obvious is that Schematron can be used together with Schematron QuickFix to develop a user guide with relatively simple means. The

ignore example shown above gives us some hints, where a QuickFix has used foreign help structures (the `d2t:ignore` attribute) to store information in the source document for the Schematron schema, to which the schema then reacted. The XSD Guide works in a similar way. In XSD there is even a separate area for such help structures: the `xs:appInfo` element can contain any elements in a foreign namespace.

As an introductory example, we use an empty XSD schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

</xs:schema>
```

Here the XSD guide gives us the Schematron message: "The XSD guide is inactive". The QuickFix "Set the XSD guide active" is available for this message. If I execute this QuickFix, I get the following result in my source document:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:annotation>
        <xs:appinfo>
            <d2t:xsdguide xmlns:d2t="http://www.data2type.de"
status="active"/>
        </xs:appinfo>
    </xs:annotation>

</xs:schema>
```

I also get a completely different Schematron message: "Please select the basic XSD design pattern. Possible values are: 'Venetian Blind' or 'Salami Slice'".

So the Schematron first checks if there is a global `xs:annotation/ xs:appinfo/d2t:xsdguide[@status = 'active']`. If this is not the case, all other Schematron checks are not displayed at all. In Schematron this works via a global variable:

```
<sch:let name="config" value="/xs:schema/xs:annotation/xs:appinfo/
d2t:xsdguide"/>
<sch:let name="status" value="($config/@status, 'inactive')[1]"/>
```

Each pattern receives an empty rule that catches all nodes when the status is inactive:

```
<sch:pattern id="mode">
```

```
<sch:rule context="node()[$status = 'inactive']"/>
```

Due to the sequential processing of the rules, no node in this pattern is checked by another rule if the status is inactive.

7.2.3. Multiple Choice

If you stick to this example, you will see that after activating the XSD Guide, a design pattern should be selected initially. Here you can see that the XSD Guide is limited to the patterns Venetian Blind and Salami Slice. Now the user has to make a multiple choice decision. In SQF there are even different ways to realize this. For this example, the simplest way was chosen: A separate QuickFix was developed for each possible answer.

```
<sch:let name="design-pattern" value="$config/@mode"/>


<sch:assert test="$design-pattern = ('venetian-blind', 'salami-slice')"
sqf:fix="mode.venetian.blind mode.element.based setGuideInActive">Please
select the basic XSD design pattern. Possible patterns are: "Venetian
Blind" or "Salami Slice".</sch:assert>
<sqf:fix id="mode.venetian.blind">
    <sqf:description>
        <sqf:title>Choose the Venetian Blind pattern.</sqf:title>
        <sqf:p>The Venetian Blind pattern will generate for each element
top-level xsd:complexType or xsd:simpleType elements.</sqf:p>
        <sqf:p>The local elements will refer to these types by type
attributes.</sqf:p>
    </sqf:description>
    <sqf:add match="$config" target="mode" node-type="attribute"
select="'venetian-blind'"/>
</sqf:fix>
<sqf:fix id="mode.element.based">
    <sqf:description>
        <sqf:title>Choose the Salami Slice pattern.</sqf:title>
        <sqf:p>The Salami Slice pattern will generate for each element
top-level xsd:element elements.</sqf:p>
        <sqf:p>The local element declarations will refer to these
elements by ref attributes.</sqf:p>
    </sqf:description>
    <sqf:add match="$config" target="mode" node-type="attribute"
select="'salami-slice'"/>
</sqf:fix>
```

Depending on which design pattern is desired, the user can execute the corresponding QuickFix and this selection is stored as a mode attribute in the `d2t:xsdguide` element. The other Schematron patterns react accordingly.

The generic QuickFixes are another possibility to send multiple choice queries to the user. They have a similar effect for the user, but can offer different response options depending on the current source document. An example of this is hidden in the XSD Guide. First we have to skip a few steps and assume that we already have the following structure:

```
<xs:complexType name="bodyType" mixed="false">
    <xs:sequence>
        <xs:element name="title"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="titleType">

</xs:complexType>
<xs:complexType name="titleType2">

</xs:complexType>
```

Here the title element declaration in the `bodyType` type has no specification of the type. So we get the error message "Please specify the type of the element title."

In addition to various QuickFixes, which all receive element declarations that have no type specification, two additional Reuse QuickFixes are offered in this special constellation:

- "Reuse the complex type 'titleType'"
- "Reuse the complex type 'titleType2'"

These two QuickFixes are created within a single `sqf:fix` element:

```
<sch:let name="name" value="@name"/>

<sqf:fix id="vb.elementType.complexReuse" use-for-each="/xs:schema/
(xs:complexType | xs:simpleType)[matches(@name, concat($name, 'Type
\d*'))]">
```

For each global complex type or simple type that matches the pattern `{$name}Type\d*`, a separate QuickFix is created.

In addition to these two multiple choice variants, the Escali plugin[7] offers another option, which is described below.

[7] https://github.com/schematron-quickfix/escali-package/tree/master/escaliOxygen

7.2.4. Input by the user

However, an XSD schema cannot be developed by multipe choice queries alone. It also requires the ability to ask the user questions that can be answered without restrictions on predefined values. An example – here we jump back to the first steps with the XSD Guide – is the selection of the root element. For the name of the root element there is no finite number of possible answers. A multiple choice query is meaningless. SQF contains UserEntries, which were introduced for exactly such a question to the user.

If the user has activated the XSD Guide and selected his design pattern, the Schematron message "You should start with a root element" is displayed. This message has a QuickFix "Define a root element name". If this QuickFix is executed, the user is asked by the UserEntry what the name of the root element should be ("Please specify the local name of your root element").

In Schematron it looks like this:

```
<sch:rule context="xs:schema" role="info">
      <sch:assert test="xs:element" sqf:fix="vb.root.define
sl.root.define">You should start with a root element</sch:assert>
      <sqf:fix id="vb.root.define">
          <sqf:description>
              <sqf:title>Define a root element name</sqf:title>
          </sqf:description>
          <sqf:user-entry name="vb.root.element.name">
              <sqf:description>
                  <sqf:title>Please specify the local name of your root
element</sqf:title>
              </sqf:description>
          </sqf:user-entry>
          <sqf:add position="last-child">
              <xs:element name="{$vb.root.element.name}"
type="{$vb.root.element.name}Type"/>
                  <!-- ... -->
          </sqf:add>
      </sqf:fix>
```

7.2.5. Mode filter by rule order and use-when condition

The XSD Guide must react differently depending on the respective context and the user's previous decisions (e.g. selection of the design pattern). Two concepts are used to make this distinction. First, as shown above in the status check, the sequential processing of the rules within a Schematron pattern is used to show or hide all rules of a pattern, depending on a global configuration.

There are patterns that were developed only for Venetian Blind and others only for Salami Slice. By convention these are distinguished by an ID prefix "vb." and "sl.". They are activated or deactivated by the first empty `sch:rule` elements:

```
<sch:let name="config" value="/xs:schema/xs:annotation/xs:appinfo/
d2t:xsdguide"/>
<sch:let name="design-pattern" value="$config/@mode"/>

<sch:let name="isSalamiSlice" value="$design-pattern = 'salami-slice'"/>
<sch:let name="isVenetianBlind" value="$design-pattern = 'venetian-
blind'"/>

<sch:pattern id="vb.content">
    <sch:rule context="node()[$status = 'inactive']"/>
    <sch:rule context="node()[not($isVenetianBlind)]"/>

        <!-- ... -->

    <sch:pattern id="sl.content">
        <sch:rule context="node()[$status = 'inactive']"/>
        <sch:rule context="node()[not($isSalamiSlice)]"/>
```

Generally, for each pattern all rules are deactivated if the XSD Guide is inactive by catching all nodes if the condition `$status = 'inactive'` is true. Additionally, for the `vb.content` pattern, all nodes are caught if the design pattern is not Venetian Blind (`not($isVenetianBlind)`).

There are also patterns that apply to all design patterns (with the ID prefix "g."). If there are QuickFixes that are only suitable for a special design pattern, the use-when condition is used:

```
<sqf:fix id="vb.root.define" use-when="$isVenetianBlind">
    <sqf:description>
        <sqf:title>Define a root element name</sqf:title>
    </sqf:description>
    <!-- ... -->
</sqf:fix>
<sqf:fix id="sl.root.define" use-when="$isSalamiSlice">
    <sqf:description>
        <sqf:title>Define a root element name</sqf:title>
    </sqf:description>
    <!-- ... -->
</sqf:fix>
```

Due to the opposing use-when conditions, only one of the QuickFixes is always visible. However, as they have the same descriptions, the user does not see any difference. The internal functionality of the QuickFixes differs, of course.

7.2.6. Complex tasks with XSLT

An important part of the XSD Guide is the description of a content model. Once the setup (activation, design pattern, root element) is defined, the following steps are repeated in recursive form:

1. Determination of a content model
2. Definition of the attributes
3. Generation of missing element or type declarations.

The most complex task is probably the content model. Here, elements can be defined in different variants of nested sequences and/or choices. Now you could display a Schematron message for each element, sequence or choice, with various options: Insert element before, insert element after, insert sequence after/before, etc. This only leads to an endless number of QuickFixes that have to be executed to create a practical content model.

Therefore, the XSD Guide supports a DTD-like syntax. The QuickFix "Edit/ Specify the content with DTD syntax" has a UserEntry "Use the usual DTD syntax to specify the content". As a value for the UserEntry, the user can now describe the content for the corresponding type or element as being used to it from DTDs and also from RELAX NG.

Example 1:

```
(head, body)
```

becomes:

```
<xs:sequence>
    <xs:element name="head"/>
    <xs:element name="body"/>
</xs:sequence>
```

Example 2:

```
(head, body+, (epilog|index)?)
```

becomes:

```
<xs:sequence>
    <xs:element name="head"/>
    <xs:element name="body" maxOccurs="unbounded"/>
    <xs:choice minOccurs="0">
        <xs:element name="epilog"/>
        <xs:element name="index"/>
    </xs:choice>
</xs:sequence>
```

A small extension of the DTD syntax also occurs:

Curly brackets can also be used to make special `minOccurs`/`maxOccurs` specifications:

```
(head, body{1:3}, (epilog | index)?)
```

becomes:

```
<xs:sequence>
    <xs:element name="head"/>
    <xs:element name="body" maxOccurs="3"/>
    <xs:choice minOccurs="0">
        <xs:element name="epilog"/>
        <xs:element name="index"/>
    </xs:choice>
</xs:sequence>
```

Of course, such conversions from a string representation to an XML structure go beyond SQF's own functions. Such tasks were solved in the XSD Guide by an XSLT function.

7.2.7. Escali plugin feature

Basically, the XSD Guide is designed to work with both SQF implementations in the Oxygen XML Editor: The Oxygen build-in implementation[8] and the Escali Oxygen Plugin[9].

The difference between the two implementations is minimal. However, there is one feature in the Escali plugin that was used here and for which there is no representation in the build-in implementation: the drop-down menus for UserEntries. Here, the Escali plugin uses a trick that is not provided by the SQF specification, but does not contradict it. In the default value for the UserEntry, a sequence is specified instead of a single value. As soon as the Escali plugin receives a sequence in the default value, it creates a drop-down menu containing the values of the sequence.

Example:

```
<sch:let name="es-impl" value="function-available('es:getPhase')"/>
<sch:let name="xsd-common-types" value="
    ('xs:string',
    'xs:boolean',
    'xs:integer',
    'xs:double',
    'xs:dateTime',
```

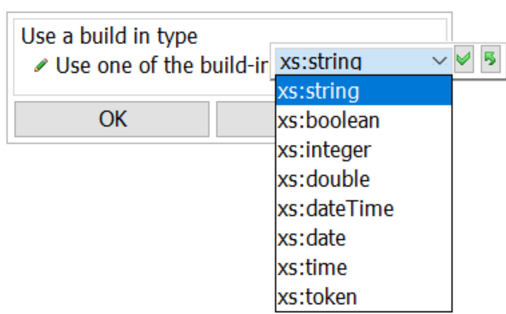
[8] http://www.oxygenxml.com

[9] https://github.com/schematron-quickfix/escali-package/tree/master/escaliOxygen

```
        'xs:date',
        'xs:time',
        'xs:token'
        )"/>
    <sch:let name="types-as-default" value="
        $xsd-common-types[if ($es-impl) then
            true()
            else
            1]
        "/>
    <!-- ... -->

    <sqf:fix id="vb.elementType.xsdtype">
        <sqf:description>
            <sqf:title>Use a build-in type</sqf:title>
        </sqf:description>
        <sqf:user-entry name="type" default="$types-as-default">
            <sqf:description>
                <sqf:title>Use one of the build-in types</sqf:title>
            </sqf:description>
        </sqf:user-entry>
        <sqf:add node-type="attribute" target="type" select="$type"/>
    </sqf:fix>
```

The `$es-impl` and `$types-as-default` variables limit the sequence of XSD types in `$xsd-common-types` to a single value only, except for the Escali plugin implementation. Otherwise, all types from the `$xsd-common-types` variable are taken over into the `$types-as-default` variable. Thus the `$type` UserEntry gets a sequence of possible values as default value for the Escali plugin. With the plugin the UserEntry looks like this:



7.2.8. Restrictions and open issues

This first version of the XSD Guide is just a proof of concept and can be extended as you like.

Known open issues are:

- A support of a target namespace is still missing completely.
- The limited use of UserEntries results in each `xs:complexType` receiving a `mixed` attribute once the content model has been determined, even if no `#PCDATA` is used. The value `false`, which the attribute gets, is not wrong, but since it is the default value of the attribute, it could actually be omitted. A clean-up mechanism is still missing here.

7.2.9. Summary and conclusions

It is important to note that the XSD Guide was only used as an example to demonstrate the basic possibilities offered by an interaction between Schematron and SQF. In the XML world, there are many other standards that have related problems and for which a similar guide might be more suitable. It should be noted that both Schematron and SQF are used for other purposes. Instead of an examination and the following correction, dummy checks are made here and the "corrections" serve rather for the structure generation.

A disadvantage is that an edited XSD schema can receive a large number of Schematron messages when the XSD Guide is active, and this could irritate a naive developer. Although most of these messages are marked with the `role` attribute as information (`info`), the corresponding elements are still highlighted. QuickFixes without Schematron messages that pop up on any node and can perform an action for that context would be desirable.

To achieve this, we suggest two possibilities: The simple method would be another misappropriation. The values for the `role` attribute are not specified by the Schematron standard. If a further value is supported here, e.g. `refactoring`, an implementation could not even display the error message, but only the corresponding QuickFixes.

A second possibility would be more conceptual in nature and would have more far-reaching consequences. Already today, independent SQF libraries can be created and embedded in Schematron schemata via `sch:include`. If one now imagines such a SQF library as an independent script, the individual QuickFixes only lack a context in which they are to be represented. If you replace the `id` attribute of the `sqf:fix` element with a `context` attribute (with the same task as the `context` attribute of the `sch:rule` element), the QuickFix would be independent. Additional conditions for providing a QuickFix, as currently provided by `sch:assert` or `sch:report`, could be implemented with `use-when` conditions.

As a result, an independent "refactoring language" would be created with minimal structural changes. But this is just an idea and needs to be discussed thoroughly.

8. Conclusion

Schematron has become a very popular language in the XML world. In the last few years, Schematron started to be used more and more and in numerous domains. One of the reasons Schematron started to be more popular is because now you have the ability to also define actions that will allow the user to correct the problem, rather than just presenting an error message. SQF has made Schematron more powerful and became a powerful language itself.

Tagdiff: a diffing tool for highlighting differences in the tagging of text-oriented XML documents

Cyril Briquet
<cyril.briquet@canopeer.org>

Abstract

Finding differences in the tagging of two XML documents can be very useful when evaluating the output of tagging algorithms. The tool described in this paper, called tagdiff, is an original attempt to take into account the specifics of text-oriented XML documents (such as the importance of lateral proximity and unimportance of ancestor/child relationships) and algorithmic tagging (lack of validity of tagged XML documents, or XML schema too relaxed to be useful), as opposed to a general purpose diffing tool such as the well-known (but XML-oblivious) diff, or as opposed to more recent tools offering comparison of XML trees, or to tools offering rich graphical user interfaces that are not suited to long lines of XML contents. In practice, the two compared XML documents are split into small typed segments that are aligned with one another and compared. These small typed segments of text or markup are aligned and printed next to one another, into two columns. Effort is made to ease visual inspection of differences.

Keywords: XML, diff, tagging, algorithm

1. Introduction

Structure-oriented XML documents have received much attention so far: *"In many applications XML documents can be treated as unordered trees – only ancestor relationships are significant, while the left-to-right order among siblings is not significant."* [10]

While XML documents are usually modeled as trees, there also exists many use cases where XML is relied upon to (semantically) tag sections of text. In text-oriented XML documents, the ancestor/child navigation axes are not so important whereas what's in immediate lateral proximity (to the left, to the right) is of utmost importance.

Finding differences between two text-oriented XML documents (typically two versions of the same text-oriented XML document resulting from the tagging of slightly different algorithms, or of subsequent algorithms in a processing sequence) is an interesting and important problem in algorithmic tagging projects, for example those using the update facility of XQuery [9], or those rely-

ing on home-made algorithms [1]. Debugging the XML tagging algorithms and understanding the linguistic surprises highlighted by unexpected tagging are not straightforward. It is thus important to be able to compare easily the output of tagged XML documents.

Diffing two XML documents with the well-known command line *diff* tool [4] (see Example 1) is not the best option because the output of this tool (totally suited for diffing files containing mostly short lines of source code) is difficult to read for XML documents. In particular, for text-oriented documents, lines may be very long and extend to a whole paragraph. Presenting differing fragments one above the other, at some distance and with lots of surrounding visual noise, also does not lend itself to easy visual inspection.

Example 1. Example: output of *diff* tool for 4 differences in a paragraph

```
< <p>Heapsort was invented by <link><person>J. W. J. Williams</person></link
> in <link><date>1964</date></link>. This was also the birth of the heap, pr
esented already by Williams as a useful data structure in its own right.</p>
---
> <p>Heapsort was invented by <link><person><b>J. W. J. Williams</b></person
></link> in <link><date><b>1964</b></date></link>. This was also the birth o
f the heap, presented already by Williams as a useful data structure in its
own right.</p>
```

Diffing two XML documents with well-known XML tools with a rich graphical user interface, such as the Oxygen XML Editor [8] and the DeltaXML XML Compare tool [3], is possible and differences are highlighted with nice colors. Comparison can be typically configured in several useful ways such as ignoring differences in certain markup types (e.g. comments, processing instructions, ...). However, the contents of the compared XML documents remain readable only as long as the lines remain conveniently short. Whenever a paragraph of the XML documents is longer than half the screen width (which is typical of many text-oriented XML documents), the overflowing contents are not displayed. Moreover, the differences are not systematically vertically aligned with one another.

Diffing XML documents is also possible with tools targeting tree-oriented XML documents (some of these tools are recent, such as the XQuery-based one presented at XML Prague 2018 based on the approach proposed by the prior X-Diff tool [10]). These tools take into account the arborescent nature of XML and allow for differences in the ordering of subtrees. However, such tools focus on the changes between contents of XML elements rather than on tags added, removed or moving around laterally.

The command line *tagdiff* tool that we describe in this paper is an original attempt at diffing text-oriented XML documents. *Tagdiff* displays the two text-oriented XML documents side-by-side, splits them into typed segments of limited

width that fit within one vertical column, and displays contexts around the differences (see Example 2, and contrast it with the previous Figure here above).

The rest of this paper is structured as follows: Section 2 describes the algorithm on which *tagdiff* is based, Section 3 discusses its implementation, Section 4 provides a preliminary evaluation, Section 5 lists related work, and Section 6 concludes the paper.

Example 2. Example: output of *tagdiff* tool for 4 differences in a paragraph

```
============================================================================

   6 .                                 =      6 .
   6 </p>                              =      6 </p>
   6 <?eoln?>                          =      6 <?eoln?>
   7 <?eoln?>                          =      7 <?eoln?>
   8 <p>                               =      8 <p>
   8 Heapsort was invented by          =      8 Heapsort was invented by
   8 <link>                            =      8 <link>
   8 <person>                          =      8 <person>
                                     <--->    8 <b>
   8 J. W. J. Williams                 =      8 J. W. J. Williams

============================================================================

   8 J. W. J. Williams                 =      8 J. W. J. Williams
                                     <--->    8 </b>
   8 </person>                         =      8 </person>
   8 </link>                           =      8 </link>
   8  in                               =      8  in
   8 <link>                            =      8 <link>
   8 <date>                            =      8 <date>

============================================================================

   8 </person>                         =      8 </person>
   8 </link>                           =      8 </link>
   8  in                               =      8  in
   8 <link>                            =      8 <link>
   8 <date>                            =      8 <date>
                                     <--->    8 <b>
   8 1964                              =      8 1964

============================================================================
   8 1964                              =      8 1964
                                     <--->    8 </b>
   8 </date>                           =      8 </date>
```

```
   8 </link>                       =      8 </link>
   8 . This was also the birth of  =      8 . This was also the birth of
   8 the heap, presented already b =      8 the heap, presented already b
   8 y Williams as a useful data s =      8 y Williams as a useful data s
   8 tructure in its own right.    =      8 tructure in its own right.
   8 </p>                          =      8 </p>
   8 <?eoln?>                      =      8 <?eoln?>
============================================================================
```

2. Algorithm

The main idea behind the algorithm on which the *tagdiff* tool is based is to segment the XML documents into small typed segments that are easy to align (between the two XML documents), easy to compare, and easy to visualize.

The algorithm is composed of five phases: (1) Diffing of the unparsed XML documents, (2) Segmentation of the parsed XML documents, (3) Visual segmentation, (4) Alignment of sequences of differing segments, and (5) Prettyprinting.

2.1. First phase: Diffing of the unparsed XML documents

The first phase consists in comparing the unparsed XML documents (the raw data, without parsing), in order to identify which sections of the two XML documents are equal and which are differing. This is a preprocessing step for the following phases, and it does not involve XML parsing.

For this first phase, the algorithm relies on a classic diffing algorithm [7] with complexity O(ND) (with N=length of the XML documents, D=length of edit script between documents) which reduces to O(N) when the differences between documents are small. This is a perfectly reasonable assumption and, even in the presence of very different XML documents, the Myers algorithm is, to the best of our current knowledge, still the way to go. (Nonetheless, see the Implementation and Evaluation Sections of this paper for important remarks about performance.)

The output of the first phase is that all sections of each XML document are typed as either equal to their counterpart, or not present in the other XML document (i.e. differing).

2.2. Second phase: Segmentation of the parsed XML documents

The second phase consists in segmenting each XML document into short, typed segments that are known (thanks to the first phase, see above) to be equal or differing with their counterparts in the other XML document.

Each XML document is modeled first as a list of XML chunks resulting from XML parsing, and then as a list of typed segments.

How does it work? Each of the XML documents goes through XML parsing. As a result of XML parsing, each XML document is first initially modeled as a list

of XML chunks (see Example 3), each chunk roughly equivalent to a DOM node without navigation axes except lateral navigation (to the left, to the right).

This modeling implies that the non-empty elements are modeled as two tag chunks each (the opening tag chunk and the closing tag chunk), and the attributes are modeled as contained in the relevant opening (or empty) tag chunk; empty elements are modeled as one empty tag chunk each.

Remark: Note that the method of parsing (i.e. SAX or DOM) is orthogonal to (i.e. independent from) the modeling into XML chunks. The modeling of the XML documents into two lists of XML chunks can be implemented based on SAX parsing or DOM parsing. End of remark.

Example 3. Each XML document is first modeled as a list of contiguous XML chunks (chunk boundaries are here represented by vertical bars)

```
|<?xml version="1.0" encoding="UTF-8"?>|<?eoln?>|<article xmlns="http://
www.tagdiff.
org/basic-markup">|<?eoln?>|<?eoln?>|<p>|In computer science, the heapsort
algorithm is a |<l
ink>|comparison-based sorting algorithm|</link>|. Heapsort can be thought o
f as an improved |<link>|selection sort|</link>|: like that algorithm, it d
ivides its input into a sorted ...| ...
```

Upon completion of the modeling of each XML document as a list of XML chunks, the algorithm iterates over both (1) the sections of each of the unparsed XML documents (known to be equal or differing thanks to the diffing algorithm in the first phase) and (2) the list of XML chunks resulting from XML parsing (whose XML node type is known from the XML parsing in this second phase). Each XML chunk is typed based on the its XML node type (text/tag/processing instruction/comment) and its equal/differing status (4 x 2 = 8, there are thus 8 types).

Types include:

- equal text,
- equal tag,
- equal processing instruction,
- equal comment,
- differing text,
- differing tag,
- differing processing instruction,
- differing comment.

Entities were resolved during the prior XML parsing of the XML document. Other items of the XML infoset (i.e. CDATA, notations, ...) are currently not taken

into account but would be modeled as chunks (and segmented) in a similar manner.

Remark: End of line characters are materialized into the segmentation as processing instructions. Two XML text chunks separated by an end of line are distinct and thus assigned to separate segments. Comments split over two or more consecutive lines are not necessarily assigned to separate segments (and ends of lines enclosed in a comment are displayed as a space character for readability purposes). End of remark.

See Example 4 for an example of our proposed segmentation of two XML documents.

Each equal-data XML chunk corresponding to multiple XML chunks in the other document is split into multiple segments. For example, in document 1 of Example 4, the text chunk |In computer science| is split into two equal-data segments, |In | and |computer science|.

Each XML chunk comprised of multiple sections of equal or differing contents (e.g. multiple equal types and multiple differing types) is typed as one segment only, and always of a differing type. It must be noted that the corresponding XML chunk in the other XML document also needs to be similarly segmented and typed as a differing type (this is trivially achieved by using the data obtained in the first phase). For example, in both documents modeled in Example 4, the <article> tag is typed in both documents as a differing tag.

Example 4. Each XML document is modeled as a list of typed segments (segment boundaries are here represented by vertical bars)

```
first XML document

|<?xml version="1.0" encoding="UTF-8"?>| equal processing instruction
|<?eoln?>| equal processing instruction
|<article xmlns="http://www.tagdiff.org/basic-markup">| differing tag
|<?eoln?>| equal processing instruction
|<?eoln?>| equal processing instruction
|<p>| equal tag
|In | equal text
|computer science| equal text
|, the | equal text
|heapsort| equal text
| algorithm is a | equal text
|<link>| equal tag
|comparison-based sorting algorithm| equal text
|</link>| equal tag
...

second XML document
```

```
|<?xml version="1.0" encoding="UTF-8"?>| equal processing instruction
|<?eoln?>| equal processing instruction
|<article book="1" volume="20" ici="2" xmlns=...| differing tag
|<?eoln?>| equal processing instruction
|<?eoln?>| equal processing instruction
|<p>| equal tag
|In | equal text
|<link>| differing tag
|<b>| differing tag
|computer science| equal text
|</b>| differing tag
|</link>| differing tag
|, the | equal text
|<link>| differing tag
|<b>| differing tag
heapsort| equal text
|</b>| differing tag
|</link>| differing tag
| algorithm is a | equal text
|<link>| equal tag
|<b>| differing tag
|comparison-based sorting algorithm| equal text
|</b>| differing tag
|</link>| equal tag
...
```

The output of the second phase consists in the two XML documents modeled in a data structure each (referred to as the segmentation of an XML document) that lists the derived segments and associates each of them (1) with its type, (2) with its character offset in the equal-data version of the XML document (i.e. the concatenation of all equal sections of the XML document, which is also the unparsed version of the XML document with all differing sections removed), as well as (3) with the line number where it's stored in the XML document.

For each of the two XML documents, the complexity of the segmentation is linear in the number of typed segments generated by the algorithm, which is at most the length of the XML document. The complexity of the second phase is thus O(N) (with N=length of the XML documents).

2.3. Third phase: Visual segmentation

The third phase consists in segmenting further the segments obtained at the end of the second phase, so that each segment can be printed on a half page (the left half for the first XML document, the right half for the second XML document). Each segment is further segmented into one or more segments (see Example 5). This splitting depends on the length of each segment: shorter, longer or even sev-

eral times longer than a half page, with the length of half page provided as a parameter to the algorithm (typically 29 characters for displaying in classic terminals).

This further visual segmentation ensures that both the visual comparison and the algorithmic comparison are straightforward, as will be seen in the following.

Example 5. XML chunks are segmented into typed segments, themselves possibly further segmented into several typed segments of limited width in order to facilitate both the algorithmic and the visual inspection of differences

```
1 <?xml version="1.0" encoding=   =      1 <?xml version="1.0" encoding=
1 "UTF-8"?>                       =      1 "UTF-8"?>
1 <?eoln?>                        =      1 <?eoln?>
2 <article xmlns="http://www.ta <--->    2 <article book="1" ici="2" lan
2 gdiff.org/basic-markup">      <--->    2 g="english" volume="20" xmlns
                                <--->    2 ="http://www.tagdiff.org/basi
                                <--->    2 c-markup">
2 <?eoln?>                        =      2 <?eoln?>
3 <?eoln?>                        =      3 <?eoln?>
4 <p>                             =      4 <p>
4 In                              =      4 In
                                 ...

4 comparison-based sorting algo   =      4 comparison-based sorting algo
4 rithm                           =      4 rithm


                                 ...

4 selection sort                  =      4 selection sort
                                <--->    4 </b>
4 </link>                         =      4 </link>
4 : like that algorithm, it div   =      4 : like that algorithm, it div
4 ides its input into a sorted    =      4 ides its input into a sorted
```

The output of the third phase is a segmentation of each XML document into smaller segments than those output by the second phase. The complexity of the third phase is thus again O(N) (with N=length of the XML documents).

2.4. Fourth phase: Alignment of sequences of differing segments

The fourth phase consists in aligning and comparing the segments produced by the third phase. The main idea consists in grouping together into sequences the segments that may be different between the XML documents, in order to significantly facilitate and reduce in size the alignments and comparisons to be performed.

The first step (of the fourth phase) thus consists in grouping together the neighboring differing segments. For example, see Example 6.

Example 6. Neighboring differing segments are grouped together

```
=============================================================================
==
example of an area of the two XML documents with differing markup and text
=============================================================================
==

   6 <link>                           =      6 <link>
                                    <--->    6 <b>
   6 in-place algorithm             <--->    6 piece of work
                                    <--->    6 </b>
   6 </link>                          =      6 </link>

=============================================================================
==
example of 3 segment sequences in the first (left-side) XML document
=============================================================================
==

sequence i (equal tag segment):
|<link>|

sequence i+1 (differing text segment):
|in-place algorithm|

sequence i+2 (equal tag segment):
|</link>|

=============================================================================
==
example of 3 segment sequences in the second (right-side) XML document
=============================================================================
==

sequence j (equal tag segment):
|<link>|

sequence j+1 (differing tag and text segments):
|<b>|
|piece of work|
|</b>|

sequence j+2 (equal tag segment):
```

```
|</link>|

===========================================================================
==
```

The second step (of the fourth phase) is as follows. Once grouped together into sequences, the segments are aligned at two levels: sequence level and then segment level.

Alignment at the sequence level:

- The sequences of equal data, always comprised of exactly one segment, are aligned (at the sequence level, which is - in the case of equal data - also the segment level) with their identical counterparts by relying on their character offsets in the equal-data version of the XML documents. Those character offsets were computed in the second phase (segmentation) of the algorithm.

  Remark: A frequently occurring special case is when a segment of equal data is split by an added tag in the other XML document. For example, see Example 7, where "In computer science" in one XML document is split into "In " and "computer science" in the other XML document. A correction must be made when aligning the segments of equal data: whenever one segment of equal data at a given character offset is a prefix of a segment of equal data at the same character offset, the latter must be split into two segments of equal data. The two new segments of equal data will then be aligned with their counterparts by the regular alignment algorithm, just like any other segments of equal data. End of remark.
- The sequences comprised of several differing segments are aligned (at the sequence level) with their possibly differing counterparts in the other XML document, based on the character offset of the nearest (i.e. previous) sequence of equal data. Intuitively and very importantly, this corresponds to aligning together what's in between parts of the XML documents that are guaranteed (as known from the first phase) to be equal, based on their character offsets in the data-equal version of the XML documents. See Example 6 and Example 8 for a visual illustration.

Example 7. Segments of equal data may not be identical (frequent special case)

```
===========================================================================
==
incorrect (on the left side): one equal text segment "In computer science"
===========================================================================
==

   4 In computer science           <--->    4 In
                                   <--->    4 <link>
                                   <--->    4 <b>
```

```
                                    <--->    4 computer science

============================================================================
==
correct (on the left side): two equal text segments (after splitting)
============================================================================
==

   4 In                               =      4 In
                                    <--->    4 <link>
                                    <--->    4 <b>
   4 computer science                 =      4 computer science

============================================================================
==
```

Alignment at the segment level: when a sequence of differing segments differs from its counterpart, there are typically "gaps" in one or both of the sequences, corresponding, for instance, to added or removed tags (see Example 8). The alignment between pairs of matching sequences is a combinatorial problem to solve for each pair of matching sequences of differing segments.

Example 8. Alignment at the segment level of two sequences of differing segments (the equal-data segments above and below are already aligned based on their offsets)

```
============================================================================
==
two sequences of segments that need alignment
============================================================================
==
    |<link>|                          =       |<link>|
    |in-place algorithm|              ?       |<b>|
    (gap)                             ?       |piece of work|
    (gap)                             ?       |</b>|
    |</link>|                         =       |</link>|
============================================================================
==
the same sequences, aligned
============================================================================
==
    |<link>|                          =       |<link>|
    (gap)                           <--->     |<b>|
    |in-place algorithm|            <--->     |piece of work|
    (gap)                           <--->     |</b>|
    |</link>|                         =       |</link>|
```

==
==

The third step (of the fourth phase) thus consists in a heuristic to find the best possible alignment between matching sequences of differing segments. We propose that a difference minimization algorithm finds the best possible alignment.

Relying on systematic recursive enumeration of possible alignments (with early pruning of unpromising solutions), the optimal alignment that minimizes the number of differences between segments of the two sequences of differing segments is chosen. Comparisons are made at the segment level, based on their contents (which are rather small due to the third phase of the algorithm). Furthermore, differences between segments are weighted by segment type, so that e.g. tags are aligned preferably with other tags, text preferably with other text, ...

An upper bound on the complexity of the fourth phase is $O(Ng^s)$ (with N=length of the XML documents, g=number of gaps=number of segments of the longest sequence minus number of segments of the shortest sequence, s=number of slots=1 plus number of segments of the shortest sequence).

However, in practice, this complexity is not going to be a problem because (1) it is a very large upper bound (computing a tighter bound would be a nice endeavor) and (2) the number of gaps and slots (for example, see Example 8 where there are 2 gaps and 2 slots) are typically small given that the alignment algorithm is applied, one at a time, to sequences of differing segments in-between segments of equal data. Moreover, again, long sequences of differing segments are found in structure-oriented XML documents rather than text-oriented XML documents. For the running example illustrating this paper, both the mean number of gaps per sequence and the mean number of slots per sequence are both around 2.

2.5. Fifth phase: Prettyprinting

The fifth phase consists in prettyprinting the differences between segments detected in the previous phases.

Differences between the two XML documents are presented to the user as a sequence of contextualized items. As already said, the two versions of each differing segment are printed next to one another, into two columns. Each difference is contextualized by printing several neighboring segments above and below, without duplication of context beyond the previous and following differences.

Differing segments (the differences) and equal segments (the contexts) are clearly identified. Line numbers are provided. See again, for example, Example 2.

The complexity of the fifth phase is O(N) (with N=length of the XML documents).

3. Implementation

The presented algorithm is implemented as a command line tool programmed in Java 8 (about 5000 lines of code, plus the various libraries used for finding differences in raw data, for XML parsing and for the XML data model).

The tool requires only the path to the two XML documents two compare. Well-formedness is enforced but validation is not performed.

Indeed, to find the differences between two XML documents using *tagdiff*, a schema is not only unneeded but could also prevent the parsing of XML documents that are non-valid. For example, in our use case (see the introductory Section of this paper), the newest of the two XML documents to compare may include non-valid markup spliced by the newest (and buggy) version of the tagging algorithm that produced it.

Diffing the unparsed versions of the XML documents in the first phase of our proposed algorithm relies on a high performance Google library (Diff Match and Patch[1]) that implements a classic diffing algorithm [7]. However, measurements of processing of actual data showed that the runtime of the Myers diffing algorithm increased very quickly with the number of differences, resulting in poor performance.

To boost performance, we optimized the use of the Myers algorithm (not the algorithm itself) as follows. The intent is to split the two XML documents to compare into relatively short sections with not too many differences that will keep the cost of the Myers algorithm to a reasonable level for each short section. The question is then to find, in the two XML documents, splitting points that are guaranteed to match. The splitting points we chose are the paragraph tags (<p> and </p>), making the assumption that no paragraph was added or removed between the two XML documents. Other splitting points (to be specified by the user as a regular expression) are possible, as long as they can be assumed to be equal between the two XML documents. The Myers algorithm is thus applied many times to the short sections in-between and within the paragraphs. And it works, as illustrated in the Evaluation section here below.

Compared with a tree-to-tree comparison, this optimization (1) is less general and requires the user to provide a bit of knowledge about the XML documents (if the default choice of paragraph tags is not applicable), but (2) has neither the memory overhead required by two complete XML trees, nor the extra work required for XML parsing and for the tree-to-tree comparison; finally, (3) it must be conceded that this solution is a bit arborescent in nature with a 3-level modeling of the XML documents (root, paragraphs, contents).

Obtaining chunks in the second phase (segmentation of the parsed XML documents) is currently done through SAX parsing and a chunk-based XML data

[1] https://github.com/google/diff-match-patch

model [1]. This could be based on another data model as long as chunks can be extracted into the appropriate in-memory data structure, or chunks could even be obtained *a posteriori* after e.g. DOM parsing. These are implementation considerations that are independent from the diffing algorithm proposed in this paper.

4. Evaluation

Descriptions of an algorithm typically include an evaluation of its performance. As a preliminary evaluation, the *tagdiff* algorithm has been tested on a small test corpus of 3 pairs of XML documents:

- a pair of small XML documents: 14 lines and about 2 kB each, 26 differences between them;
- a pair of medium XML documents: about 1000 lines and 115 kB each, 743 differences between them;
- another pair of medium XML documents: about 4000 lines and 500 kB each, 3638 differences between them.

The *tagdiff* algorithm has been applied to these XML documents, 3 times (for reliability of results), on 2 different test computers equipped, respectively, with a high-end CPU (Intel Core i7) and a commodity one (Intel Celeron). Figure 1 shows that, for the high-end CPU, the runtime is a fraction of a second for the small XML documents and a few seconds for the medium documents. It also illustrates that performance is much better with the optimization for the use of the Myers diffing algorithm described here above in the Implementation Section. The results are also about twice as fast on the high-end CPU, both with or without the optimization of the diffing algorithm.

Figure 2 gives (for the Intel Core i7 CPU) the breakdown of runtimes for the algorithm components: (1) diffing of the raw documents, (2) XML parsing, (3) segmentation and (4) alignment (the combinatorial optimization problem). Table 1 gives (for the Intel Core i7 CPU) the actual runtimes. Two observations can be made.

Firstly, XML parsing dominates the runtime for the small XML documents and, while growing less than linearly with the XML document sizes (not shown on the figure but available in the table: 142 ms, 472 ms, 1235 ms, respectively), quickly becomes a fraction of the total runtime.

Secondly, the runtime of the diffing library increases with the XML document sizes. While initially very small, it rapidly becomes a fraction of the total runtime but does not become overly large.

Table 2 gives (for the Intel Core i7 CPU) the runtimes for the second pair of medium XML documents for a variable number of differences. The XML document containing differences is 3858 lines long and we compared 3 versions of it: about 1 difference every line, about 1 difference every 2 lines, and about 1 differ-

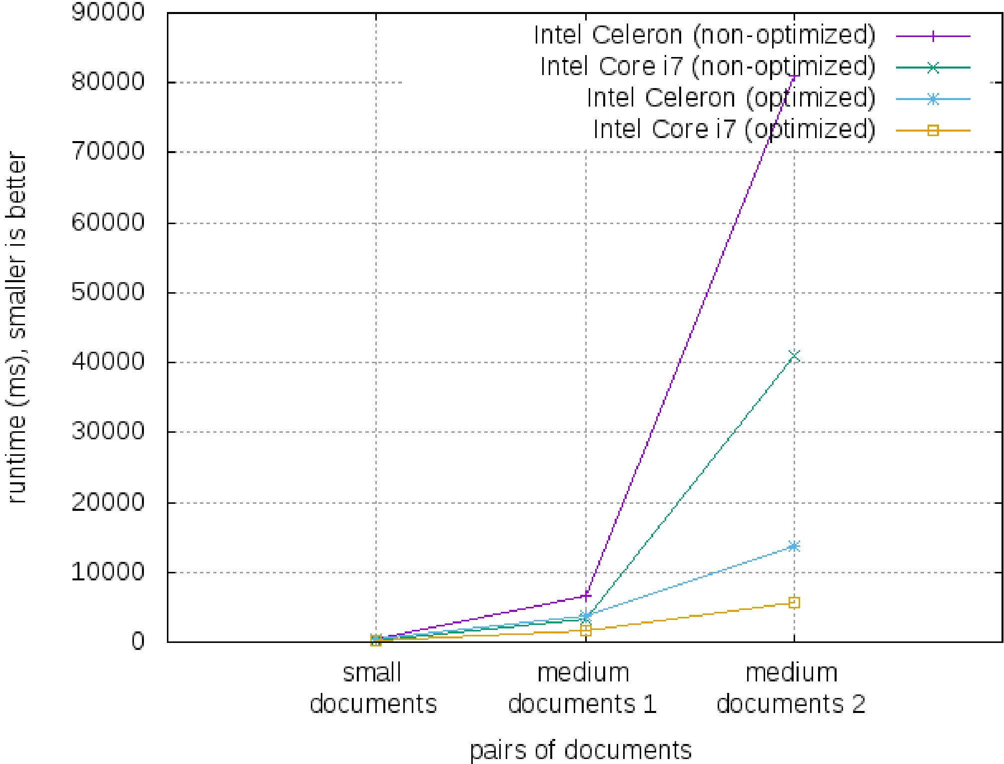


Figure 1. *tagdiff* runtimes

ence every 4 lines (the latter is the one described in the rightmost columnn of the previous Table). This ought to illustrate whether or not the diffing component decreases significantly with the number of differences between the XML documents. This is not the case, confirming that our proposed optimization of the use of the diffing library limits the growth of the runtimes with the number of differences.

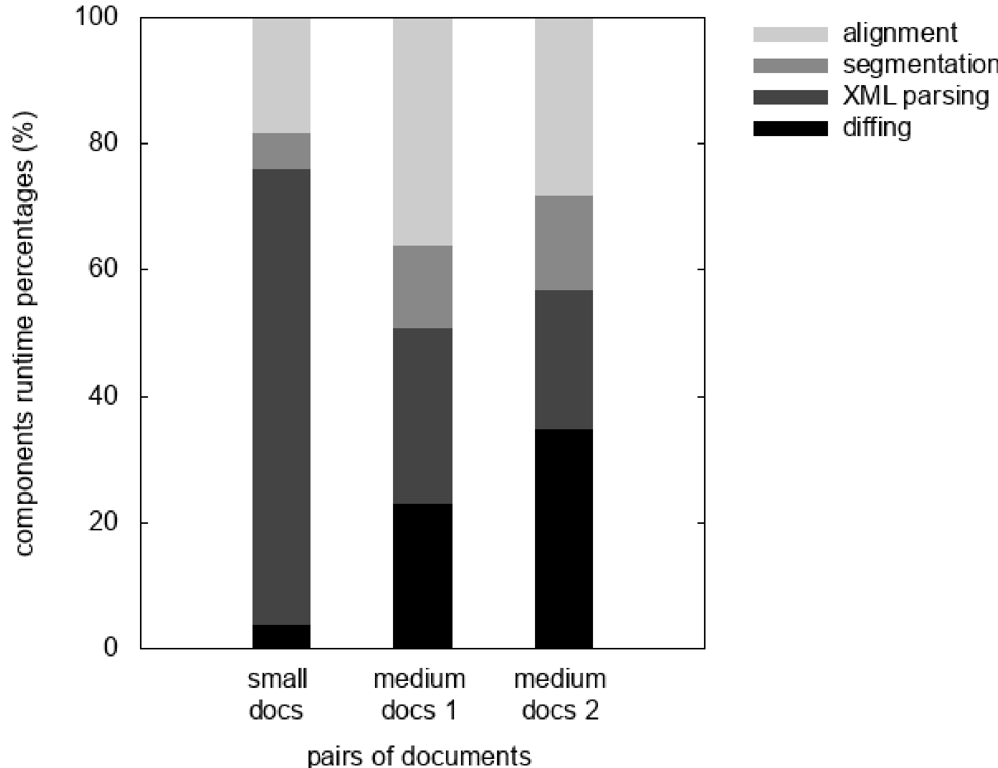


Figure 2. ***tagdiff*** **components runtimes breakdown**

Table 1. ***tagdiff*** **components runtimes (fixed number of differences)**

Runtimes (ms)	small docs	medium docs 1	medium docs 2
diffing	8	378	1987
XML parsing	142	472	1235
segmentation	12	220	882
alignment	35	593	1630
total	197	1663	5734

Table 2. ***tagdiff*** **components runtimes (varying number of differences)**

Runtimes (ms)	medium docs 2, 1066 d. (29%)	medium docs 2, 1912 d. (53%)	medium docs 2, 3637 d. (100%)
diffing	1142	1315	1987
XML parsing	973	1020	1235
segmentation	664	560	882
alignment	1302	1567	1630
total	4082 (71%)	4464 (78%)	5734 (100%)

Finally, some preliminary measurements of memory usage. Memory usage was measured (using VisualVM[2]) for the processing of the three pairs of XML documents. For most of the execution, actual heap size was about, 10 MB, 30 MB and 90 MB, respectively. During the very last phase of the execution (prettyprinting, as described in the previous section), memory usage spiked a bit due to the instantiation of lots of strings to generate the output of *tagdiff*.

5. Related work

5.1. Diffing algorithms

The Myers diffing algorithm [7] is used in the first phase of our proposed algorithm, but not in the fourth phase. Indeed, the Myers diffing algorithm operates on untyped strings, not on typed segments, and thus does not provide alignment where there are differences. For example, see again Example 8 where |in-place algorithm| would be aligned with |<b>| instead of |piece of work| if the Myers diffing algorithm were used also for differing segments.

Other prior diffing algorithms (e.g. Xyleme project [2], X-Diff algorithm [10]) focus on structure-oriented XML documents rather than text-oriented XML documents. Differences between XML documents are reported in terms of modified contents of subtrees instead of tags inserted and deleted. Prior diffing tools also study performance and memory consumption but give only limited attention to the visualization of results.

5.2. Command line tools

The classic *diff* tool [4] outputs differences line by line, which quickly becomes unreadable as the line length becomes longer than half the screen width (see Example 2).

The classic *diff* tool [4] with the -y option activated prints the two documents side-by-side but does not segment long lines into segments of maximum fixed width, and does not output differences as typed segments, each on a dedicated line.

The classic *diff* tool [4] with the -u option activated provides data to patch an existing file, which the *tagdiff* tool doesn't. This is not a goal of *tagdiff*, which is intended for human users, not for processing by another software (though the output of a command line tool can be provided to other command line tools, of course).

The DiffMk tool [5] relies on the Myers diffing algorithm [7] and on tree-to-tree comparison. On small test cases, it has slightly better performance than *tagdiff*

[2] https://visualvm.github.io/

but doesn't do segmentation and alignment. It highlight differences by inlining the differing segments within the text common to the two XML documents.

5.3. GUI tools

Tools with a rich graphical user interface, such as the Oxygen XML Editor [8] and the DeltaXML XML Compare tool [3], offer diffing with a variety of options and filters (such as exact comparison, compare by word, ignore certain items of the XML infoset, ...). However, the differences between the compared XML documents remain readable only as long as the lines remain conveniently short. Whenever a paragraph of the XML documents is longer than half the screen width (which is typical of many text-oriented XML documents), the overflowing contents are not displayed. Moreover, the differences are not systematically vertically aligned with one another.

There also exists (web-based) tools, such as the W3C HTML diff service [6], that highlight differences by inlining the differing segments within the text common to the two XML documents, at the expense of discarding differences in markup.

5.4. Usability

In terms of usability, *tagdiff* is as simple as it gets as it is a command tool that expects as arguments only the paths to the two XML documents to compare. As discussed in the previous paragraphs, we also believe that *tagdiff* does a good job of facilitating visualization of differences between text-oriented XML documents.

Responsiveness is also a component of usability. In terms of performance, it is however too early to draw conclusions. Additional evaluation and benchmarking are needed but, as seen in the previous section, the number of differences have the biggest impact on the Myers diffing algorithm [7]. In addition to this, the cost of XML parsing, which is amortized for large documents, has a big impact for the diffing of small XML documents.

6. Conclusion

This paper presents an original attempt to provide a tool that facilitates the visualization of differences in the tagging of two text-oriented XML documents.

The tool described, *tagdiff*, segments the chunks of the XML documents (both text and markup are individually segmented) and then compares aligned sequences of typed segments:

- detection of differences and alignment of equal segments are done by a classic diffing algorithm [7], applied to small sections of the XML documents (in a slightly arborescent fashion) to obtain good performance,

- alignment of differing segments is done by a combinatorial difference minimization algorithm (with pruning) that is applied a large number of times, but to typically small data, thus keeping its runtime under control.

  Differences are then contextualized and prettyprinted.

7. Acknowledgements

We would like to thank the anonymous reviewers for their helpful suggestions that helped improve this paper.

Bibliography

[1] C. Briquet, P. Renders and E. Petitjean. *A virtualization-based retrieval and update API for XML-encoded corpora.* In Proc. Balisage, Montréal, Québec, 2010.

[2] G. Cobéna, S. Abiteboul, A. Marian. *Detecting Changes in XML Documents.* In Proc. Int. Conf. on Data Engineering, San Jose, 2002.

[3] *DeltaXML.* [online]: https://www.deltaxml.com/

[4] *Diff.* In Wikipedia, the Free Encyclopedia. [online]: https://en.wikipedia.org/wiki/Diff

[5] *DiffMk.* [online]: https://sourceforge.net/projects/diffmk/

[6] *HTML diff service.* [online]: https://services.w3.org/htmldiff

[7] E. Myers. *An O(ND) Difference Algorithm and Its Variations.* In Algorithmica 1, 2, 1986.

[8] *Oxygen XML Editor.* [online]: https://www.oxygenxml.com/

[9] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton and J. Siméon. *XQuery Update Facility 1.0.* W3C Recommendation [online]: http://www.w3.org/TR/xquery-update-10/

[10] Y. Wang, D. DeWitt and J. Cai. *X-Diff: An Effective Change Detection Algorithm for XML Documents.* In Proc. Int. Conf. on Data Engineering, Bangalore, India, 2003.

Merge and Graft: Two Twins That Need To Grow Apart

Robin La Fontaine
DeltaXML
<robin.lafontaine@deltaxml.com>

Nigel Whitaker
DeltaXML
<nigel.whitaker@deltaxml.com>

Abstract

Software developers are familiar with merge, for example pulling together changes from one branch into another in a version control system. Graft (also known as 'cherry pick') is a more selective process, pulling changes from selected commits onto another branch. These two processes are often implemented in the same way, but they are not the same, there are subtle but important differences.

Git and Mercurial have different ways to determine the correct ancestor when performing a merge operation. Graft operations use yet another way to determine the ancestor. In the built-in line-based algorithm, the underlying diff3 process is then used. This diff3 algorithm accepts non-conflicting changes and relies on the user to resolve conflicts. We will examine the details of this process and suggest that the symmetrical treatment that diff3 uses is appropriate for merge but not necessarily optimal for the graft operation.

We will look at examples of tree-based structures such as XML and JSON to show how different approaches are appropriate for merge and graft, but the arguments presented apply to the merging of any structured content. Treating structured documents and data as tree-structured rather than just as a sequence of lines provides more intelligent merge and graft results, and enables rules to be applied to give improved results, thus reducing time-consuming and tedious manual conflict resolution.

We examine how these improvements can be integrated into version control systems, taking Git as an example, and suggest developments in their architecture that will make them even more powerful development tools.

Keywords: XML, JSON, Merge, Graft, Cherry-pick, Git

1. Introduction

Merging files across branches in a version control system can be a very time-consuming process. Automation is a great help, of course, but has its limitations, some of which are fundamental and some of which are due to the merge tools not fully understanding the source file structure.

One fundamental limitation is that it is not always possible to automate the propagation of changes because conflicts may occur. A conflict is a situation where accepting one change is not compatible with accepting another change – a choice needs to be made and so user intervention is needed. User intervention obviously slows down the process because it can take significant time to understand a conflict sufficiently to resolve it. A high level of expertise is needed to resolve conflicts and this is therefore an expensive process, not to mention being a very tedious job requiring sustained high levels of concentration.

It is worth mentioning that it is possible to resolve all conflicts automatically if a precedent order is defined, i.e. the changes in one branch will always take priority. We will discuss this in more detail later.

Merge tools are, usually, line-based tools that treat the files as a sequence of lines but they have no understanding of any syntactic structure that may be present. The result is often surprisingly good but can go very wrong in some situations, for example when the text has been re-formatted. Errors in the result can also occur because, for example, a new start tag insertion has been accepted but the corresponding end tag, perhaps several hundreds of lines further down the file, has been rejected – this type of error may be difficult to avoid, and very time consuming to correct.

Our concern in this paper is with processing XML, which has a tree structure, but the discussion also applies to other file types that can be converted into XML [1]. One example of this is JSON which also has a tree structure and can be converted into XML for processing [2] and then the result transformed back into JSON.

For XML, the merge process begins by merging all three source files into a single file where common data is shared and the differences between the three files are represented in a well-defined way. We call this intermediate merged file a delta file. Rules can then be applied to the delta file so that any changes are processed to generate a resultant merged file, ideally one that does not contain any conflicts. If there are conflicts, these can be represented in XML for downstream conflict resolution.

It is this ability to represent the three files in one, and the definition of rules to process the changes, that enables us to refine our understanding of the process and so identify subtle but important differences between the regular merge process and the graft process. This paper explores these differences as a step towards

refining automated merge/graft processing to improve it and so save time and effort.

2. Terminology

First we will define our terminology and describe the merge and graft processes. Graft is also often known as 'cherry pick' but we will use the shorter term here.

Given two branches, each with two versions:

- Branch P with versions P0, P1 and P2
- Branch Q with versions Q0, Q1 and Q2

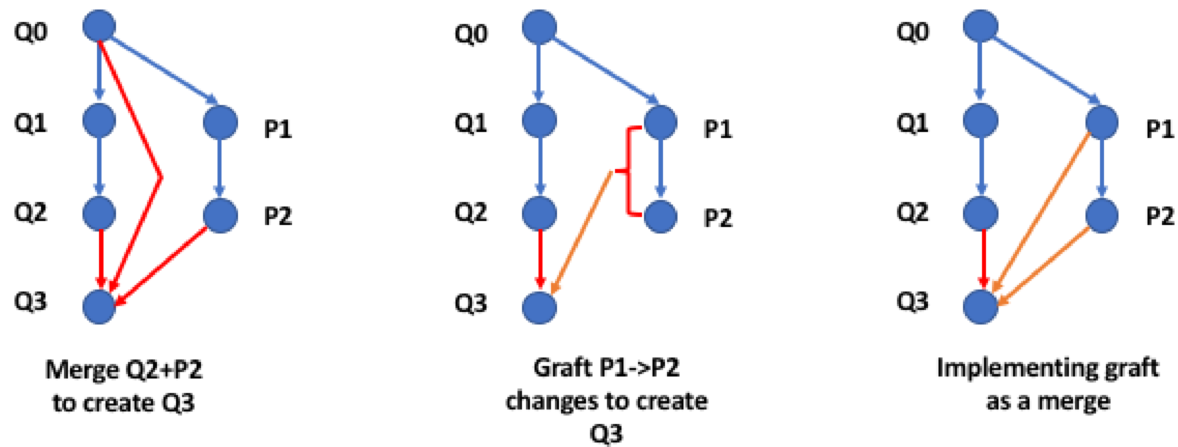


Figure 1. Merge and Graft Operations

The figure above shows the classic merge operation on the left. Graft, in the centre, cherry-picks the changes from P1 to P2 and applies this to Q2 to produce Q3. The figure on the right is the way that graft is implemented as a merge, so that P1 is considered the ancestor.

For merge, P0=Q0 in other words P1 and Q1 have a common ancestor.

The merge process is to create Q3 such that all the changes between Q0 and P2 and all the changes between Q0 and Q2 are represented in the changes between Q0 and Q3. This may result in conflicts that need to be resolved.

For graft, P0 may not equal Q0, but we want to propagate the changes between two selected commits on the same branch, such as P1 and P2, onto another branch, for example the one ending Q2, to create a new commit Q3.

These processes are not the same, and we will present XML and JSON examples to illustrate in detail how using the same algorithm for both does not always give optimal results.

The graft process is therefore to create Q3 such that all the relevant changes between P1 and P2 are represented in the changes between Q2 (the 'target' data

set) and Q3. A change is only relevant if the data involved in the change is present in the target data set. This may also result in conflicts that need to be resolved, but typically there are fewer (or no) conflicts caused by graft than those caused by merge - for reasons that are described later.

3. A word about conflicts

As noted in the Introduction, it is possible to resolve all conflicts automatically if a precedent order is defined, i.e. the changes in one branch will always take priority. Conflicts are not usually resolved in this way because it does not always give the ‘right’ result simply because some conflicts require user review to understand why they have occurred and how best to correct them. The ‘right’ result may not be the acceptance of one change or the other but rather some form or merging of the changes. This is the case both for source code merge and for document merges.

There may be situations when a precedent order can be defined and if so time can be saved because some, if not all, conflicts can be resolved automatically. The ability to do this is a desirable feature in a merge application.

4. How do merge and graft differ?

Regular three-way merge is a symmetrical process in that the two branches being merged are considered to have equal status. We do not want a change in one to override a change in the other without some user checking. Therefore we want conflicts to be identified. With tree structured data conflicts can be more complex, for example a sub-tree is deleted in one branch but it has also been changed in the other branch.

The use case for merge is well understood in version control systems, i.e. the requirement to merge changes from a branch back into the trunk or master version. The use case for graft is not so obvious, and it is worth considering use cases where graft is really what is required rather than merge. Within a version control system, graft is appropriate for 'cherry picking' changes between two versions on one branch to propagate these changes across to another branch. For source code, we might want to apply a bug fix but no other changes to another branch. It is also the appropriate process for keeping related sets of data synchronised in terms of changes made to either set.

Consider an example related to Master Data Management (MDM). We have a set of contact information (name, address, phone etc.) for members of an organisation. A smaller list or subset might be held for a local office, but how can this be kept up to date with changes to the master set of members? Regular three-way merge is not appropriate because a change to a member who is not in this office would result in an apparent conflict (amended in master set and deleted in local

office set). Graft would work because this change would not be relevant so would be ignored. What about deletion? If a member leaves, then that member is deleted in the master set and both graft and merge would propagate this deletion to the local office set. But if a member moves from the local office to another office, then the deletion from the local office set would similarly propagate to the master set - which is not what is needed. This is an example where a 'safe graft' or 'graft without deletions' would be needed - a variation on the normal graft that requires different rules. Although this applies to a specific use case, this use case supports the argument that different use cases require different rules.

For regular merge, the appropriate common ancestor file is identified and each branch has the same relationship with it in that it has been derived from it. Therefore differences between the ancestor and each branch are 'real' changes that need to be considered. In some situations, the appropriate ancestor file is unambiguous, but in other more complex merge scenarios it may be necessary for more complex processing to determine what is appropriate. Further discussion of this is outside the scope of this paper.

For graft, on the other hand, we do not have a common ancestor. Therefore the graft situation is not symmetrical. We have two files from one branch and we are trying to apply the changes between these to some other file – typically a file that is very similar but there may be significant differences.

The principles can be illustrated with a simple example. This is a very simple example and seems rather trivial but the simplicity is intended to clarify the actual changes from the detail of the data.

Consider some file that contains contact information, identified by a name. To keep this very simple, we have used a string "v1" to represent the data rather than more realistic details of address, phone etc. Therefore "v2" represents some change to that data.

Given two branches, each with two versions:

- Branch P with versions P0, P1 and P2
- Branch Q with versions Q0, Q1 and Q2

We now consider and example to show that the implementation of graft as a merge does not always produce the desired results.

We started with P0 and Q0 containing these contacts: John, Mike, Anna. We will look at it as JSON, which is more compact than XML, but of course it could be XML.

Example 1. P0=Q0

```
{
  "John": "v1",
  "Mike": "v1",
```

```
  "Anna": "v1"
}
```

We took these and added David into P1: John, Mike, Anna, David

Example 2. P1:

```
{
  "John": "v1",
  "Mike": "v1",
  "Anna": "v1",
  "David": "v1"
}
```

We deleted John and changed details for Mike and David in P2: Mike has been changed and this is denoted by a new value "v2", Anna is unchanged, and David is also changed, so again we indicate this simply with a new value "v2"

Example 3. P2:

```
{
  "Mike": "v2",
  "Anna": "v1",
  "David": "v2"
}
```

The above describes what happened on the P branch. Now we can look at the changes made on the Q branch. First, we changed details for Anna in Q1: John, Mike, Anna is now "v2".

Example 4. Q1:

```
{
  "John": "v1",
  "Mike": "v1",
  "Anna": "v2"
}
```

We add a new contact, Jane, and change John in Q2: John is now "v2", Mike, Anna is now "v2", Jane

Example 5. Q2:

```
{
  "John": "v2",
  "Mike": "v1",
  "Anna": "v2",
```

```
  "Jane": "v1"
}
```

A regular three-way merge of these two branches would result in a merged Q branch:

Q3 would have a conflict for John who has been deleted in P but changed in Q, Mike has value "v2", Anna likewise is "v2", Jane is unchanged and there is a conflict for David who appears to have been deleted in Q2 but at the same time has been changed in P.

Example 6. 3-way merge

```
P1                    P2                  Q2                  Q3: 3-way merge
{                   {                   {                   {
  "John": "v1",                           "John": "v2",       ! CONFLICT
  "Mike": "v1",       "Mike": "v2",       "Mike": "v1",       "Mike": "v2",
  "Anna": "v1",       "Anna": "v1",       "Anna": "v2",       "Anna": "v2",
  "David": "v1"       "David": "v2"                           ! CONFLICT
                                          "Jane": "v1"        "Jane": "v1"
}                    }                  }                   }
```

On the other hand, if we want to perform a graft then we want to propagate the changes between P1 and P2 onto the Q branch. The changes between P1 and P2 are to Mike and David, and John is deleted. However, as David does not appear in Q2 we cannot apply that change – it is not a conflict, it is just a change that is not relevant.

So the result is Q3: Mike has value "v2", Anna has value "v2", Jane is unchanged.

Example 7. Graft

```
P1                    P2                  Q2                  Q3: Graft
{                   {                   {                   {
  "John": "v1",                         "John": "v2",
  "Mike": "v1",     "Mike": "v2",       "Mike": "v1",         "Mike": "v2",
  "Anna": "v1",     "Anna": "v1",       "Anna": "v2",         "Anna": "v2",
  "David": "v1"     "David": "v2"
                                        "Jane": "v1"          "Jane": "v1"
}                   }                   }                   }
```

Can we get the same result using a regular three way merge with a precedent on changes, e.g. that changes in Q override changes in P? In this case the deletion of David overrides the change in David in the P branch. However, the change in John in Q overrides the deletion in P, so we would get:

Example 8. 3-way merge, Q priority

```
P1                   P2                  Q2                  Q3: 3-way merge,
                                                                 Q priority
{                    {                   {                   {
  "John": "v1",                            "John": "v2",       "John": "v2",
  "Mike": "v1",        "Mike": "v2",       "Mike": "v1",       "Mike": "v2",
  "Anna": "v1",        "Anna": "v1",       "Anna": "v2",       "Anna": "v2",
  "David": "v1"        "David": "v2"
                                           "Jane": "v1"        "Jane": "v1"
}                    }                   }                   }
```

Similarly, a merge with changes in P overriding changes in Q would leave David in the result which differs from the graft result, as shown below.

Example 9. 3-way merge, P priority

```
P1                   P2                  Q2                  Q3: 3-way merge,
                                                                 P priority
{                    {                   {                   {
  "John": "v1",                            "John": "v2",
  "Mike": "v1",        "Mike": "v2",       "Mike": "v1",       "Mike": "v2",
  "Anna": "v1",        "Anna": "v1",       "Anna": "v2",       "Anna": "v2",
  "David": "v1"        "David": "v2"                           "David": "v2"
                                           "Jane": "v1"        "Jane": "v1"
}                    }                   }                   }
```

This shows that the rules for merge are different from the rules for graft. They may appear to be only different in a subtle way but the differences in the result of a complex merge are significant. So too would be the time saved in getting the automated result to be correct.

Some readers may say that the result of the graft is not exactly what they want because they do not want to ever have any data deleted, they only want additions to be propagated. Such a requirement might arise, for example, when we do not want master data deleted or changed. This is shown below.

Example 10. Graft: Additions only

```
P1                   P2                  Q2                  Q3: Graft:
                                                                 Additions
                                                                 only
{                    {                   {                   {
  "John": "v1",                            "John": "v2",       "John": "v2",
  "Mike": "v1",        "Mike": "v2",       "Mike": "v1",       "Mike": "v1",
  "Anna": "v1",        "Anna": "v1",       "Anna": "v2",       "Anna": "v2",
  "David": "v1"        "David": "v2"
```

```
                                          "Jane": "v1"      "Jane": "v1"
  }                  }                  }                  }
```

The conclusion is that the merge or graft rules need to be controlled in different situations, and indeed may need to be controlled in a different way across an XML tree. The ability to define the rules according to the use case is an advantage even if the standard definitions work in most situations.

We summarise the above results in the following table.

Table 1. Changes and their different interpretation for Merge and Graft

P1	P2	Q2	Merge interpretation	Graft interpretation	Comment
"John": "v1"		"John": "v2"	Deleted by P2, changed by Q2 - conflict	Deleted	We do not know if Q2 has changed this because we do not have its ancestor
"Mike": "v1"	"Mike": "v2"	"Mike": "v1"	Changed by P2	Changed	
"Anna": "v1"	"Anna": "v1"	"Anna": "v2"	Changed by Q2	No change to apply	
"David": "v1"	"David": "v2"		Changed by P2, deleted by Q2 - conflict	Data not present in target so cannot apply change	
		"Jane": "v1"	Added by Q2	No relevant changes to apply	
	"Martin"="v1"		Added by P2	Added	

The table above shows how the same situation is interpreted in a different way by merge and graft. Similarly, we can create a table to show a number of possible types of graft operation.

Table 2. Variants of Graft

P1	P2	Q2	'Regular' graft interpretation	'Additions only' graft interpretation	'No deletions' graft interpretation
"John": "v1"		"John": "v2"	Deleted	No change	No change
"Mike": "v1"	"Mike": "v2"	"Mike": "v1"	Changed	No change	Changed
"Anna": "v1"	"Anna": "v1"	"Anna": "v2"	No change to apply	No change	No change
"David": "v1"	"David": "v2"		Data not present in target so cannot apply change	No change	No change
		"Jane": "v1"	No relevant changes to apply	No change	No change
	"Martin"="v1"		Added	Added	Added

These different flavours of graft are appropriate in different situations. For example if the Q branch is the master data then we might not want deletions to be applied. On the other hand if the P branch is the master one then we may want to apply all changes and deletions. In tree-structured data the situation is more complex because we may want to apply different rules at different places in the tree.

5. Advantages of XML for rule-based conflict resolution

Merge or graft rules can potentially become very adept at handling different situations, reducing the need to resolve conflicts by inspection.

It is very difficult to apply useful rules to line-based merges. There are two reasons for this. First, most data is structured in some way, for example the tree structure of XML and JSON. Source code is also structured and can typically be represented as a tree structure. Any line-based representation will not reflect this structure and so it is not possible to represent sensible rules. Secondly, the standard representations of change to text files, for example the output from diff or the diff3 format, do not easily support rule-based processing.

XML does provide significant advantages in this area. First, it is possible to merge several documents into one and represent the similarities and differences

in XML. Given this representation, rules can then be defined to process the merged document to generate a result, so different sets of rules will generate different results according to the requirements. A major benefit of XML or JSON markup is that the conflicts can be *addressed* using mechanisms such as XPath [10] or JSON Pointer [9]. This enables much more powerful automated or rule-based processing. XPath and XSLT provide a well-defined language to specify the rules and execute them.

For example, consider the merge of a Maven POM file (a system for software build configuration). In the POM file we describe dependencies of the code being built and in a number of circumstances we have found version conflicts in POM dependencies. We could identify these conflicts with an XPath: `/project/dependencies/dependency/version` and could say that for these conflicts we will use the version in the current or HEAD branch, or perhaps, assuming the dependency has an upwardly compatible API, we take the highest version.

It is the addressability of tree-based conflicts that introduces these automation possibilities. If conflict resolution rules can be applied in the merge driver then fewer conflicts would need to be presented to the user to deal with manually in a merge tool. We could expect that rules could be created to improve the conflict processing for a number of formats which are manually or semi-automatically created using developer tools such as IDEs, including Apache Ant, Maven POMs and XSLT.

Looking further ahead, other structured data could be converted into XML, and then back again, as shown with 'Invisible XML' 1. This approach opens up the opportunity to apply the intelligent, tree-structured XML merge to other types of data, then after application of rules the result can be converted back from XML into the original format. This is how the above JSON was processed 3.

6. Integration with Git Merge

We have argued here that 'merge' is not a fixed process, rather it changes according to different needs and the nature of the data. We now move on to discuss how we might go about implementing this for Git.

The merge (and also graft) process of Git involves a number of components, these are:

- merge scenario
- merge driver
- merge tool

The 'merge scenario' is responsible for looking at all of the files and directories with an understanding of moves and renames, matching up the corresponding files, determining the appropriate ancestor and calling the merge driver on triples of files. In some cases the scenario will determine that a full merge is unnecessary

and may, for example, perform a fast-forward merge. It is also possible to specify a scenario such as 'mine' that produces a result that takes all of the files on a certain branch. In these cases it is not really a full merge and the merge driver may not be invoked.

The 'merge driver' receives three files corresponding to the ancestor and the two branches, loads the content into memory and is responsible for aligning their content and identifying any conflicts. Using a return code, it signals to the invoking code whether there are any conflicts. This usually reports a message to the user, often using a line starting with a capital "C" character. The diff3 merge driver in Git represents conflicts using a textual line based format consisting of marker lines using angle-bracket, equals or plus characters. The **git config** command can be used to configure different merge drivers and the `.gitattributes` file can then select between them using the filenames or extensions of the files being merged.

The user typically needs to resolve any conflicts in each file before the merge operation can be completed and committed. It is possible to take the file with the markers produced by the driver and resolve the conflicts by editing that output in a text editor and then reporting that the file has been resolved. A 'merge tool' provides a graphical user interface to automate the conflict resolution process, often allowing the user to select content from one of the branches or possibly the ancestor for each of the conflicting regions in the file.

There are two common usage or interaction patterns we have found relating to the use of merge drivers and merge tools:

- The merge driver produces the line-based conflict markers and then the merge tool reads the result file from the driver, interprets the markers and provides the user with selection capabilities based on this interpretation. Merge tools which take this approach include MS Visual Studio Code [5] and TkDiff [4].
- The merge tool, when it is invoked, is supplied with the filename of the driver result, but also the names of the original inputs to the merge driver. It can then re-merge the inputs and perhaps base its user interface on internal data structures from its own merge algorithm. Examples of tools which re-merge and do not seem to use the driver results include: Araxis Merge [7], P4Merge [6], and OxygenXML [8].

Given an enhanced, tree-based, three-way merge algorithm it is possible to integrate it into the Git merge process as either a merge driver or merge tool. We believe that the best approach is to integrate as a merge driver for reasons that we will summarise next.

6.1. Avoiding conflict confusion

The merge driver and merge tool should identify the same conflicts, i.e. behave in a consistent way. When processing XML with a line-based algorithm (such as

diff3), changes such as those to attribute order might cause a conflict in the merge driver. In many workflows a conflicting file would cause the merge tool to be invoked in order to resolve the conflict. But if the merge tool then uses a tree-based XML or JSON aware algorithm this would not identify these apparent conflicts and the file may not even have any conflicts present. The unnecessary invocation of the merge tool may cause confusion for the user.

6.2. Improved non-conflicting results

A tree-based merge algorithm which is XML or JSON aware would normally produce well-formed XML or JSON results. However this is not true of a line based merge such as diff3, where the result may have mismatched element tags for example. These bad results will not necessarily be associated with a conflict - the mismatched tag may be non-conflicting. If the tree-aware algorithm is only used in the merge tool, it may never be invoked unless there is a conflict and it is therefore possible for bad results to go unnoticed.

An algorithm with a better understanding of the data and its semantics can make better alignment decisions. Again in non-conflicting situations it makes sense to have this better alignment performed at the merge driver stage.

6.3. Simpler software design

The separation of the merge algorithm and a conflict resolving GUI can lead to simpler software design. It may be that merge tools find the textual markers insufficient for their needs and can provide a better experience by re-running a merge algorithm, but the merge architecture would be simpler if this was not necessary. This would avoid duplicated code and reduce the processing and IO required. In the following section we will look at this issue in greater detail.

7. Representing merge conflicts

It is possible to have an XML or JSON aware merge algorithm and then describe conflicts with diff3 compatible line-based markers discussed earlier. However, we found some limitations to this approach:

- If fine-grained textual changes are shown to the user on individual lines this can interfere with the original code or data layout and the result of the merge may be less familiar and need reformatting. Figure 2 shows one of our experiments in this area using Visual Studio code. It illustrates how an attribute conflict can be reformatted to work well with the diff3 markers and Figure 3 shows the resolved result.
- The line based conflict format represents a linear sequence of conflicts. However tree based merge algorithms can have nested changes and (with an n-

way or octopus merge) nested conflicts. Consider the case where one branch modifies a single word in a paragraph and in the other merge branch that paragraph is deleted. If the deletion is accepted then there is no need to descend further into that subtree to deal with any nested changes or conflicts.

- For XML, if a conflicting insertion of a start tag is accepted, then the corresponding end tag should also be included, but there is no way to specify these linked edits in diff3. Similarly for JSON, it is not simple to ensure that the separating commas are included correctly.

Figure 2. XML attribute conflict in Microsoft Visual Studio Code

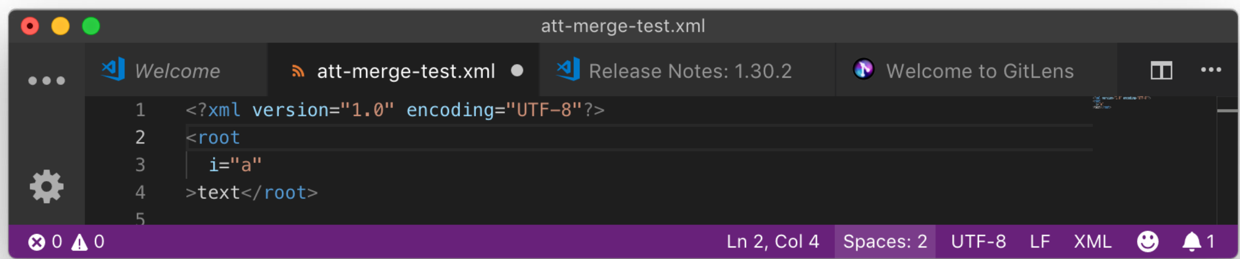


Figure 3. Resolved attribute conflict in Microsoft Visual Studio Code

These limitations, together with the limited number of merge tools which handle the diff3 format, have made us consider alternative approaches. Representing the conflict as XML or JSON markup allows us to overcome the limitations above and could allow better communication between the merge driver and merge tool. The intention would be to provide a rich enough change representation so that there

is no need to re-run the merge algorithm. Merge tool applications can then take advantage of different merge drivers, and users can choose their preferred merge tool.

Some merge tools handle the data as lines of text, others are more aware of the type of data, e.g. OxygenXML understands the structure of XML or JSON. This leads to different requirements for the way that conflicts are communicated from the merge driver to the merge tool. A fully XML aware merge driver could communicate detailed XML semantics to an XML aware merge tool, but could only convey more limited information to a merge tool that understands only line based text files.

Conflict markers from diff3 may have line numbers but these are not helpful in free-format XML or JSON structures. On the other hand, users do care about layout for source code and for some JSON or XML data and this should be preserved. There is no simple solution to how best to communicate between more semantically intelligent merge drivers and the requirements and capabilities of various merge tools.

8. Conclusions

We have shown how merge and graft differ and argued that a single merge process is not appropriate to cover both scenarios. This distinction can be taken further to show that there are not just two but many different types of merge, each useful and appropriate for different use cases.

The ability to represent structured data and documents as tree-based enables us to provide improved merge and graft operations. The use of XML, with XPath and XSLT, provides essential support to the development of a rule-based system that provides merge and graft for different use cases.

We have examined how such improvements can be integrated into Git and suggested developments in its architecture that will enable more versatile and appropriate merge and graft operations. The saving in time when dealing with large, complex documents and data is self-evident, but the advantage to users of reducing the time that needs to be spent on the very tedious task of conflict resolution is an even greater incentive to pursue this.

References

[1] *Data Just Wants to Be Format-Neutral*[1], Steven Pemberton, CWI

[2] *Transforming JSON using XSLT 3.0*[2] Michael Kay, Saxonica

[1] http://www.xmlprague.cz/day2-2016/#data
[2] http://www.xmlprague.cz/day2-2016/#jsonxslt

[3] *JSON Compare Online Preview - DeltaXML https://www.deltaxml.com/json-client/*

[4] *tkdiff project*[3]

[5] *Visual Studio Code*[4]

[6] *Helix P4Merge and Diff Tool*[5]

[7] *Araxis Merge*[6]

[8] *Oxygen XML Editor*[7]

[9] *RFC: 6901 JavaScript Object Notation (JSON) Pointer*[8], Internet Engineering Task Force (IETF)

[10] *XML Path Language (XPath) 3.1*[9], W3C

[3] https://sourceforge.net/projects/tkdiff/
[4] https://code.visualstudio.com
[5] https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge
[6] https://www.araxis.com/merge/
[7] https://www.oxygenxml.com/xml_editor.html
[8] https://tools.ietf.org/html/rfc6901
[9] https://www.w3.org/TR/2017/REC-xpath-31-20170321/

The Design and Implementation of FusionDB

Adam Retter
Evolved Binary
<adam@evolvedbinary.com>

Abstract

FusionDB is a new multi-model database system which was designed for the demands of the current Big Data age. FusionDB has a strong XML heritage, indeed one of its models is that of a Native XML store.

Whilst at the time of writing there are several Open Source and at least one commercial Native XML Database system available, we believe that FusionDB offers some unique properties and its multi-model foundation for storing heterogeneous types of data opens up new possibilities for cross-model queries.

This paper discusses FusionDB's raison d'être, issues that we had to overcome, and details its high-level design and architecture.

1. Introduction

FusionDB, or Project Granite as it was originally known, was conceived in the spring of 2014. Ultimately it was born out of a mix of competing emotions - frustration, excitement, and ambition. The frustration came from both, the perceived state of Open Source NXDs (Native XML databases) at that time, and commercial pressures of operating such systems reliably at scale. The overall conclusion being that they each had several critical issues that were not being addressed over a prolonged period of time, and that they were rapidly being surpassed on several fronts by their newer NoSQL document database cousins. The excitement came from witnessing an explosion of new NoSQL database options, including many document oriented database systems, which offered varying consistency and performance options, with a plethora of interesting features and query languages. Whilst the ambition came from believing that the community, i.e.: the users of NXD systems, deserved and often wanted a better option, and that if it needed building, then we both could and, more importantly, should build it!

In 2014, we had over 10 years experience with an Open Source NXD, eXist-db, both contributing to its development, and helping users deploy and use it at a variety of scales. Well aware of its strengths and weaknesses in comparison to other NXDs and non-XML database systems, we set out to identify in a more scientific manner the problems faced by the stakeholders; ourselves as the develop-

ers of the NXD, our existing users, and those users which we had not yet attracted due to either real or perceived problems and missing features.

In the remainder of this section we set out what we believe to be the most important issues as identified by eXist-db users and developers. Whilst we do make comparisons between BaseX, eXist-db, MarkLogic and other NoSQL databases, we place a particular focus on eXist-db as that is where our experience lies. It should be remembered that each of these products has varying strengths and weaknesses, and that all software has bugs. Whilst one might interpret our discussion of eXist-db issues as negative, we would rather frame it in a positive light of transparent discussion, yes there are issues, but if we are aware of them, then ultimately they can be fixed. There are many eXist-db users operating large sites who are able to cope with such issues, just as there are with BaseX, MarkLogic, and others.

1.1. Issues Identified by Users

First, through either direct reports from existing and potential users, or acting as a proxy whilst deploying solutions for users, we identified the following issues:

1. Stability

   Under normal operation the NXD could stop responding to requests. As developers, we identified two culprits here, 1) *deadlocks* caused by the overlapping schedules of concurrent operations requiring mutually exclusive access to shared resources, and 2) read and write contention upon resources in the system, whereby accessing certain resources could cause other concurrent operations to stall for unacceptably long periods of time.

2. Corruption

   When the system stopped responding to requests, it had to be forcefully restarted. Often this revealed or led to corruption of the database. As developers, we identified both a lack of the Consistency and Durability properties of ACID (Atomicity, Consistency, Isolation, and Durability) semantics, which also affected the ability to recover safely from a crash (or forceful restart).

3. Vertical Scalability

   Adding CPUs with more hardware threads did not cause the performance of the database to scale as expected. As developers, we identified many bottlenecks caused by contention upon shared resources.

4. Horizontal Scalability

   There was no clustering support for sharding large data sets or splitting resource intensive queries across multiple machines.

5. Operational Reliability

   There was no support for a mirrored system with automatic failover in the event of a hardware failure.

6. Key/Value Metadata

   There was no facility to associate additional key/value metadata with documents in the database. Many users would store the metadata externally to the document, either in a second "metadata document", or an external (typically) SQL database, later combining the two at query time. Unfortunately, this lacks atomic consistency, as updates to both the metadata and documents are not guaranteed to be in sync. As developers, we identified that a key/value model which is atomically consistent with a document model would be advantageous.

7. Performant Cross-reference Queries

   The system supported indexes for `xml:id` XML identifiers and references within and across documents. However, more complicated XML Markup Languages such as DocBook, TEI, and DITA, may use much more complex linking schemes between nodes where composite key addressing information is expressed within a single attribute. As developers, we identified that for queries spanning documents with such complex references, more advanced data models and index structures could be used to improve performance.

1.2. Issues Identified by Developers

Both through those issues raised by users, and our experience as developers, we identified a number of technical issues with eXist-db that we believed needed to be solved. Each of those technical issues fell into one of three categories - Correctness, Performance, and Missing Features.

It is perhaps worth explicitly stating that we have ordered these categories by the most important first. Correctness has a higher priority than Performance, and Performance has a higher priority than developing new features. There is little perceived benefit to a database system which is fast, but fast at giving the wrong results!

1.2.1. Correctness

1. Crash Recoverable

   If the database crashes, whether due to a forced stop, software error, or the host system losing power, it should always be possible to restore the database to a previously consistent state.

   A lack of correctness with respect to the implementation and interaction of the WAL (Write Ahead Log) and Crash Recovery process needs to be addressed.

2. Deadlock Avoidance

   In a number of scenarios, locks within the database on different types of resources are taken in different orders, this can lead to a predictable yet avoidable deadlock between two or more concurrent operations.

A lack of correctness with respect to the absolute order in which locks should be obtained and released by developers on shared resources needs to be addressed.

3. Deadlock Detection and Resolution

   With dynamic queries issued to the database by users on an ad-hoc basis, it is prohibitively expensive (or even impossible) to know all of the shared resources that will be involved and how access to them from concurrent queries should be scheduled. Deadlock Avoidance in such scenarios is impossible, as it requires a deterministic and consistently ordered locking schedule.

   With the ad-hoc locking of resources to ensure consistency between concurrent query operations, deadlocks cannot be prevented.

   To ensure the correct and continual functioning of the system, deadlocks must not cause the system to stop responding. If pessimistic concurrency with locking is further employed, a deadlock must be detected and resolved in some fashion which allows forward progress of the system as a whole. An alternative improvement would be the employment of a lock-free optimistic concurrency scheme.

4. Transaction Isolation

   With regards to concurrent transactions, Isolation (the *I* in ACID) [1], is one of the most important considerations of a database system and deeply affects the design of that system.

   Also equally important, is a clear statement to users about the available isolation levels provided by the database. Different user applications may require varying isolation levels, where some applications, such as social media, may be able to tolerate inconsistencies provided by weaker isolation levels, accounting or financial applications often require the strongest isolation levels to ensure consistency between concurrent operations.

   For example, BaseX provides clear documentation of how concurrent operations are scheduled [10]. It unfortunately does not explicitly state its isolation level, but we can infer from its scheduling that it likely provides the strictest level - *Serializable*. Likewise, MarkLogic dedicates an entire chapter of their documentation [14], to the subject, a comprehensive document that unexpectedly does not explicitly state the isolation level, but likely causes one to infer that *Snapshot Isolation* is employed; a further MarkLogic blog post appears to confirm this [15].

   The exact isolation level of eXist-db is unknown to its users and likely also its developers. Originally eXist-db allowed *dirty-reads* [2], therefore providing the weakest level of transaction isolation - *Read-Uncommitted*. Several past attempts [3][5][6][7] have been made at extending the lock lease to the transaction boundary, which would ensure the stronger *Read-Committed* or *Repeatable-Read* level. Unfortunately, those past attempts were incomplete, so we can only

infer that eXist-db provides at least *Read-Uncommitted* semantics, but for some operations may offer a stronger level of isolation.

At least one ANSI (American National Standards Institute) ACID transaction isolation level must be consistently supported, with a clear documented statement of what it is and how it functions.

5. Transaction Atomicity and Consistency

   A transaction must complete by either, *committing* or *aborting*. Committing implies that all operations within the transaction were applied to the database as though they were one unit (i.e. atomically), and then become visible to subsequent transactions. Aborting implies that no operation within the transaction was applied to the database, and no change surfaces to subsequent transactions.

   Unfortunately, the transaction mechanism in eXist-db is not atomic. If an error occurs during a transaction, it will be the case that any write operation prior to the error will have modified the database, and any write operation subsequent to the error will not have modified the database, the write operations are not undone when the transaction aborts!

   There is no guarantee that a transaction which aborts in eXist-db will leave the database in a logically consistent state. Unless a catastrophic failure happens (e.g. hardware failure), it is still possible although unlikely, for an aborted transaction to leave the database in a physically inconsistent state. Recovery from such physically inconsistent states is attempted at restart by the database Recovery Manager.

   Transactions must be both Atomic and Consistent. It should not be left as a surprise for users running complex queries, that if their query raises an error and aborts, to discover that some of their documents have been modified whilst others have not.

1.2.2. Performance

1. Reducing Contention

   Alongside the shared resources of Documents and Collections that the users of NXDs are concerned with, internally there are also many data structures that need to be safely shared between concurrent operations.

   For example, concurrent access to a Database Collection in eXist-db is effectively mutually exclusive, meaning that only a single thread, regardless as to whether it is a reader or writer, may access it. Likewise, the same applies to the paged storage files that eXist-db keeps on disk.

   To improve vertical scaling we therefore need to increase concurrent access to resources. The current liberal deployment of algorithms utilising coarse-grained locking and mutual exclusion need to be replaced, either with single-

writer/multi-reader locking, or algorithms that use a finer-grained level of locking, or where possible, non-blocking lock-free algorithms.

2. System Maintenance

   There is occasionally the need to perform maintenance tasks against the database system, such as creating backups or reindexing documents within the database.

   Often such tasks require a consistent view of the database, and so acquire exclusive access to many resources, which limits (or even removes) the ability of other concurrent queries and operations to run or complete until such maintenance tasks finish.

   For example, in eXist-db there are two backup mechanisms. The first is a best effort approach which will run concurrently with other transactions, but does not guarantee a consistent snapshot of the database. The second will wait for all other transactions to complete, and then will block any other transaction from starting until it has completed, this provides a consistent snapshot of the database, but at the cost of the database being unavailable whilst the backup is created. Another example is that of re-indexing, which blocks any other Collection operation, and therefore any other query transaction.

   Such database operations should not cause the database to become unavailable. Instead, a *version snapshot* mechanism should be developed, whereby a snapshot that provides a point-in-time consistent view of the database can be obtained cheaply. Such maintenance tasks could then be performed against a suitable snapshot.

1.2.3. Missing Features

1. Multi-Model

   The requirement from users for Document Metadata informs us that we also need the ability to store *key/value model* data alongside our documents.

   Conversely, the requirement from users for more complex linking between documents appears to us to be well suited to a *graph model*. Such a graph model, if it were available, could be used as a query index of the connections between document nodes.

   If we disregard *mixed-content*, then JSON's rising popularity, likely driven by JavaScript and the creation of Web APIs [16], places it now heavily in-demand. Many modern NoSQL document databases offer JSON (JavaScript Object Notation) [11][12][13] document storage. Undoubtedly an additional JSON *document model* would be valuable.

   Neither eXist-db nor BaseX have multi-model support, although both have some limited support for querying external relational-models via SQL (Structured Query Language), and JSON documents in XQuery [17][18]. Berkeley

DB XML offers atomic access to both key/value and XML document models [19]. MarkLogic offers both graph and document models natively [20].

It is desirable to support key/value, graph, and alternative document models such as JSON, to compliment our existing XML document model.

2. Clustering

   With the advent of relatively cheap off-the-shelf commodity servers, and now to a more extreme extent, Cloud Computing, when storage and query requirements dictate it should be possible to distribute the database across a cluster of machines.

   Should any machine fail within the cluster, the entire database should still remain available for both read and write transactions (albeit likely with reduced performance). Should the size of the database reach the ceiling of the storage available within the cluster, it should be possible to add more machines to the cluster to increase the storage space available to the cluster. Likewise, if the machines in the cluster are saturated by servicing transactions on the database, adding further machines should enable more concurrent transactions.

   Ideally, we want to achieve a shared-nothing cluster, where both data and queries are automatically distributed to nodes within the cluster, thus achieving a system with no single point of failure.

2. Design Decisions

Due to both our technical expertise and deep knowledge of eXist-db, and our commercial relationships with organisations using eXist-db, rather than developing an entirely new database system from scratch, or adopting and enhancing another Open Source database, we decided to start by forking the eXist-db codebase.

Whilst we initially adopted the eXist-db codebase, as we progressed in the development of FusionDB we constantly reevaluated our efforts against three high-level objectives, in order of importance:

1. Does a particular subsystem provide the best possible solution?
2. We must replace any inherited code whether from eXist-db or elsewhere that we cannot trust and or/verify to operate correctly.
3. We would like to maintain eXist-db API compatibility where possible.

2.1. Storage Engine

The storage engine resides at the absolute core of any database system. For disk based databases, the task of the storage engine is to manage the low-level reading and writing of data from persistent disk to and from memory. Reads from disk

occur when a query needs to access pages from the in-memory buffer pool that are not yet in memory, writes occur when pages from the in-memory buffer pool need to be flushed to disk to ensure durability.

eXist-db has its own low level storage engine, which combines the usually segregated responsibilities of managing in-memory and on-disk operations. eXist-db's engine provides a B+ tree with a paged disk file format. Originally inherited from dbXML's B+ tree implementation in 2001 [9][8], and is still recognisable as such, although to improve performance and durability it has received significant modifications, including improved in-memory page caching and support for database logging.

After an in-depth audit of the complex code forming eXist-db's BTree and associated Cache classes which make up its storage engine, we concluded that we could not easily reason about its correctness, a lack of unit tests in this area further dampened confidence. In addition, due to the write-through structure of its page cache, we identified that concurrent operations on a single B+ tree were impossible and that exclusive locking is required for the duration of either a read or write operation.

Without confidence in the storage engine of eXist-db, we faced a fundamental choice:

- Write the missing unit tests for eXist-db's storage engine so that we may assert correct behaviour, hopefully without uncovering new previously unknown issues. Then re-engineer the storage engine to improve performance for concurrent operations, add new tests for concurrent operation, and assert that it still passes all tests for correctness.

- Develop a new storage engine which offers performant and concurrent operation, with clean and documented code. A comprehensive test suite would also need to be developed which proves the correctness of the storage engine under both single-threaded and concurrent operation.

- Go shopping for a new storage engine! With the recent explosion of Open Source NoSQL databases, it seems a reasonable assumption that we might be able to find an existing well-tested and trusted storage engine that could be adapted and reused.

2.1.1. Why we opted not to improve eXist-db's

As eXist-db's storage engine is predominantly based on a dated B+ tree implementation, and not well tested, we felt that investing engineering effort in improving this would likely only yield a moderate improvement of the status quo. Instead, we really wanted to see a giant leap in both performance and foundational capabilities for building new features and services.

Considering that within the last year at least one issue was discovered and fixed that caused a database corruption related to how data was structured within a B+ tree [4], it seemed likely that further issues could also surface.

Likewise, whilst the B+ tree is still fundamental and relevant for database research, hardware has significantly advanced and now provides CPUs with multiple hardware threads, huge main memories, and faster disk IO in the form of SSDs (Solid State Disks). Exploiting the potential performance of modern hardware requires sympathetic algorithms, and recent research has delivered newer data structures derived from B-Trees. Newer examples include the B^{link} tree [21] which removes read locks to improve concurrent read throughput, Lock-Free B +Tree [22] which removes locks entirely to reduce contention and improve scalability under concurrent operation, Buffer Trees [24] and Fractal Trees [25] which perform larger sequential writes to improve linear IO performance by coalescing updates, and the Bw-Tree [23] which both eschews locks to improve concurrent scalability and utilises log structuring to improve IO.

Given these concerns and access to newer research, we elected not to improve eXist-db's current storage engine.

2.1.2. Why we opted not to build our own

Whilst we possess the technical ability, the amount of engineering effort in producing a new storage engine should not be understated.

Of utmost importance, when producing a new storage engine is ensuring correctness, i.e. that one does not lose or corrupt data. Given that the storage engine of eXist-db evolved over many years and may still have *correctness* issues, and that a search quickly reveals that many other databases also had correctness issues with their storage engines, that in some cases took years to surface and fix [26] [27] [28] [29], we have elected not to develop a new storage engine.

In line with our organisations philosophy of both, not re-inventing a worse wheel, and gaining from contributing to open source projects as part of a larger community, we believe our engineering resources are best spent elsewhere by building upon a solid and proven core, to further deliver the larger database system features which are of interest to the end users and developers.

2.1.3. How and why we chose a 3rd-party

Having decided to use an existing storage engine to replace eXist-db's, we initially started by looking for a suitable Open Source B+ Tree (or derivative) implementation written in Java. We wanted to remain with a 100% Java ecosystem if possible to ease integration. We had several requirements to meet:

- Correctness

  We must be able to either explicitly verify the correctness of the storage engine, or have a high degree of confidence in its correct behaviour.

- Performance

  The new storage engine should provide the same or better single-threaded performance than eXist-db's B+ tree. Although we were willing to sacrifice some single-threaded performance for improved concurrent scalability with multi-threading.

- Scalability

  As previously discussed in Section 2.1, eXist-db's B+ tree only allows one thread to either read or write, and due to this it cannot scale under concurrent access. The new storage engine should scale with the number of available hardware threads.

Initially, we studied the storage engines of several other Open Source databases written in Java that use B+ Trees, namely: Apache Derby, H2, HSQLDB, and Neo4j. Whilst each had a comprehensive and well tested B+ Tree implementation backed by persistent storage, there were several barriers to reuse:

- Tight Integration - each of the B+ tree implementations were tightly integrated with the other neighbouring components of their respective database.
- No clean interface - there was no way to easily just reuse the B+ tree implementation without adopting other conventions, e.g. configuration of the hosting database system.
- Surface area - to reuse an existing B+ Tree would have meant a trade-off, where we either: 1) add the entire 3rd-party database core Jar as a dependency to our project, allowing for easy updates but also adding significant amounts of unneeded code, or 2) we copy and paste code into our project which makes keeping our copy of the B+ Tree code up-to-date with respect to upstream updates or improvements a very manual and error prone task.

Succinctly put, these other database systems had likely never considered that another project might want to reuse their core storage engine, and so were not designed with that manner of componentisation in mind.

A second route that we considered, was looking for a very lean standalone storage engine implemented in Java that could be reused within our database system. We identified MapDB as a potential option, but soon discounted it due to a large number of open issues around performance and scalability [30].

Having failed to find an existing suitable Java Open Source option for a storage engine, we needed to broaden our search. We started by examining the latest research papers on fast key/value database systems, and reasoning that databases are often considered as infrastructure software and therefore more often than not written in C or C++, we removed the requirement that it must be implemented in Java. We also broadened our scope from B+ tree like storage, to instead requiring that whatever option we identified must provide excellent performance when

dealing with the large number of keys and values that make up our XML documents, including for both random access and ordered range scans.

From many new possibilities, we identified three potential candidates that could serve as our storage engine. Each of the candidates that we identified were considered to be mature products with large userbases, and stable due to being both open source and having large successful software companies involved in their development and deployment.

- LMDB (Lightning Memory-Mapped Database Manager)

  LMDB offers a B-Tree persistent storage engine written in C. It was originally designed as the database engine for OpenLDAP.

  LMDB provides full ACID semantics with *Serializable* transaction isolation. Through a Copy-on-Write mechanism for the B-Tree pages and MVCC (Multi-Version Concurrency Control), it provides read/write concurrency and claims excellent performance; readers can't block writers, and writers can't block readers, however only one concurrent write transaction is supported. One distinguishing aspect is that the code-base of LMDB is very small at just 6KLOC, potentially making it easy to understand. The small codebase is achieved by relying on the memory mapped file facilities of the underlying operating system, although this can potentially be a disadvantage as well if multiple processes are competing for resources.

  There is no built-in support for storing heterogeneous groups of homogeneous keys, often known as *Columns* or *Tables*. LMDB also strongly advises against long running transactions, such read transactions can block space reclamation, whilst write transactions block any other writes. Another consideration, although of less concern, is that LMDB is released under the OpenLDAP license. This is similar to the BSD license, and arguably an open source license in good faith, however in the stricter definition of "Open Source" the license does not comply with the OSI's (Open Source Initiative) OSD (Open Source Definition) [31].

- ForestDB

  ForestDB offers a novel HB+-Trie (Hierarchical B+-Tree based Trie) persistent storage engine written in C++11 [32]. It was designed as a replacement for CouchBase's CouchStore storage engine. The purpose of the HB+-Trie is to improve upon the B+Tree based CouchStore's ability to efficiently store and query variable-length keys.

  ForestDB provides ACID semantics with a choice of either *Read Uncommitted* or *Read Committed* transaction isolation. Through an MVCC and append only design, it supports both multiple readers and multiple writers, readers can't block writers, and writers can't block readers, however because synchronization between multiple writers is required it is recommended to only use a single writer. The MVCC approach also supports database snapshots, this

could likely be exploited to provide stronger *Snapshot Isolation* transactions and online database backups.

ForestDB through its design intrinsically copes well with heterogeneous keys, however if further grouping of homogeneous keys is required it also supports multiple distinct *KV store* across the same files. Unlike LMDB, ForestDB because of its append only design requires a compaction process to run intermittently depending on write load, such a process has to be carefully managed to avoid blocking write access to the database. ForestDB is available under the Apache 2.0 license which meets the OSI's OSD, however one not insignificant concern is that ForestDB seems to have a very small team of contributors with little diversity in the organisations that are contributing or deploying it.

- RocksDB

  Initially, we identified LevelDB from Google but at that time development appeared to have stalled, we then moved our attention to RocksDB from Facebook which was forked from LevelDB to further enhance its performance and feature set. Facebook's initial enhancements included multi-threaded compaction to increase IO and reduce write stalls, dedicated flush threads, universal style compaction, prefix scanning via Bloom Filters, and Merge (read-modify-write) operators [36]. RocksDB offers an LSM-tree persistent storage engine written in C++14. The purported advantage of the LSM tree is that writes are always sequential, both when appending to the log and when compacting files. Sequential writes, especially when batched, offer performance advantages over the random write patterns which occur with B+Trees. The trade-off is that reads upon an LSM tree require a level of indirection as the index of more than one tree may need to be accessed, although Bloom filters can significantly reduce the required IO [33] [34].

  RocksDB provides only some of the ACID properties, Durability and Atomicity (via Write Batches). Write batches act as a semi-transparent layer above the key value storage that enable you to stage many updates in memory, reads first access the write batch and then fall-through to the key value store, giving *read-your-own-writes* and therefore a primitive for building isolation with at least *Read Committed* strength [35]. Unfortunately, the developer has to manually build Consistency and Isolation into their application via appropriate synchronisation[1]. Through MVCC and append only design, it supports both multiple readers and multiple writers. Like ForestDB, the MVCC approach also supports database snapshots, which likewise could be

[1]At the time that RocksDB was adopted for FusionDB's storage engine, RocksDB offered no transactional mechanisms and therefore we developed our own. RocksDB now offers both pessimistic and optimistic transactional mechanisms

exploited to enable a developer to provide *Snapshot Isolation* strength transactions and online database backups.

RocksDB offers *Column Families* for working with heterogeneous groups of homogeneous keys, each Column Family has its own unique configuration, in-memory and on-disk structures, the only commonality is a shared WAL which is used to globally sequence updates. Each Column Family may be individually configured for specific memory and/or IO performance, which offers the developer a great deal of flexibility in tuning the database for specific datasets and workloads. In addition, due to the global sequencing, updates can be applied atomically across column families, allowing a developer to write complex read and update operations across heterogeneous sets of data.

Like ForestDB, RocksDB runs a background compaction process to merge storage files together, this could potentially lead to write stalls, although the multi-threaded approach of RocksDB eases this, it still has to be carefully managed. One concern is that RocksDB is available under the BSD-3 clause license with an additional patent grant condition, this is a complex issue, whilst the BSD license certainly meets the OSI's OSD, the patent grant is proprietary and does not.[2].

From the three potential candidates, each of which could technically work out very well, we ultimately selected RocksDB.

We quickly discounted ForestDB because of three concerns:

- Its' almost sole adopter seems to be CouchBase. This appears to be confirmed by its small GitHub community of contributors and users.
- We felt that we did not need its main advantage, of excellent performance for variable length keys. eXist-db (ignoring extended indexes) uses seven different key types for the data it needs to store, however within each key type, the key length variation is usually just a few bytes.
- There was no existing Java language binding, unlike LMDB's lmdbjava and RocksDB's RocksJava.

The choice between LMDB and RocksDB was not an easy one. Initially, we experimented by replacing just eXist-db's persistent DOM store with RocksDB, we added a layer of storage engine abstraction between eXist-db and RocksDB so that we could more easily replace RocksDB with LMDB or another storage engine should RocksDB not meet out requirements.

The reasons that we chose RocksDB over LMDB or any other storage engine were not just technical. RocksDB is the result of a team of highly skilled engineers, having first been built at Google and then advanced by Facebook (and oth-

[2]RocksDB was later relicenced by Facebook as a dual-licensed Open Source project, available under either GPL 2.0 or Apache 2.0.

ers), the storage engine is involved in almost every interaction that a user makes with the Facebook websites (facebook.com, instagram.com, and messenger.com), as of the third quarter of 2018, Facebook had 2.27 billion monthly active users. From this, we can likely assume that RocksDB has been thoroughly battle tested in production and at scale. Whilst LMDB is used in a handful of important infrastructure projects such as - OpenLDAP, Postfix, Monero, libpaxos, and PowerDNS [37], RocksDB is used by some of the largest web companies including - AirBnb, LinkedIn, NetFlix, Uber, and Yahoo [38]. LMDB seems to be predominantly developed by Symas Corporation, whereas RocksDB whilst led by Facebook has a larger number of diverse contributors. RocksDB also appears to have a much more rapid pace of development, with new features and bugfixes appearing frequently, in contrast LMDB appears to have stabilised with little new development.

Technically, RocksDB offered a much richer feature set that any other option we evaluated, a number of which we felt would be well suited for our use case. Many of the keys used within a key type by eXist-db are very uniform with common prefixes, this is highlighted when storing large XML documents with deeply nested levels, as the DLN (Dynamic Level Numbering) node identifier encoding that it employs will produce long strings with common prefixes. RocksDB offers both *Prefix Compression* and a *Prefix Seek API*, the compression ensures that common prefixes are only stored once [39] which reduces disk space and therefore IO, whilst the prefix seek builds bloom filters of key prefixes to enable faster random lookup and scan for keys with common prefixes [40]. eXist-db provides an update facility which is similar to XQuery Update, but where the updates are applied immediately to the data, in eXist-db these updates are not isolated to the transaction, RocksDB provides a *WriteBatchWithIndex* feature, which allows you to stage updates in-memory to the database, reading from this batch allows you to *read-your-own-writes* without yet applying them to the database, we recognised that this would allow us to provide stronger isolation for both eXist-db's XQuery Update equivalent and its various other updating functions. There are also many other RocksDB features that we are making use of such as atomic commit across column families, which we would otherwise have had to build ourselves atop another storage engine such as LMDB.

Since our initial prototyping with RocksDB in 2014 and subsequent wholesale adoption in 2015, it has since also been adopted as the primary storage engine by a number of other NoSQL databases with varying data models - Apache Kafka (event stream), ArangoDB (document, graph, and key/value), CockroachDB (tabular i.e. SQL), DGraph (graph), QuasarDB (time-series), SSDB (key/value), and TiKV (key/value). RocksDB storage engines have also been developed as replacements for the native storage engines of Apache Casandra (wide column store), MongoDB (document i.e. JSON), and MySQL (tabular i.e. SQL) [38] [41].

2.2. ACID Transactions

Having decided that FusionDB must support full ACID semantics, as well as the technical implementation we must also have a clear statement of our transaction isolation level. This enables developers building their applications with FusionDB to understand how concurrent transactions on the database will interact with each other, what constraints this places on the type of applications that FusionDB is best suited for, or whether they need to add any additional synchronisation within their application to support stronger isolation semantics.

Historically, there were four levels of transaction isolation, as defined by ANSI, each of which is expressed in terms of possible phenomena *dirty-read*, *fuzzy-read*, and *phantom*, that may occur when concurrent transactions operate at that level. From weakest to strongest these are: *Read Uncommitted*, *Read Committed*, *Repeatable Read*, and *Serializable*. Often users expect or even desire the strongest level, Serializable, but as this can require a great deal of synchronization between concurrent transactions, it can have a serious performance impact, and few database systems offer this option, and typically even then not as the default. To work around the dichotomy of needing to provide Serializable semantics *and* excellent performance, additional transaction levels have been developed in recent years. These are not as strong as Serializable, but exhibit less, or other, potentially acceptable phenomena, these include - *Cursor Stability*, *Snapshot Isolation*, and *Serializable Snapshot Isolation* [1] [42].

Table 1. Isolation Types Characterized by Possible Anomalies Allowed

Isolation level	**P0 Dirty Write**	**P0 Dirty Read**	**P4C Cursor Lost Update**	**P4 Lost Update**	**P2 Fuzzy Read**	**P3 Phantom**	**A5A Read Skew**	**A5A Write Skew**
Read Uncommitted	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
Read Committed	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
Repeatable Read	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible

Isolation level	P0 Dirty Write	P0 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5A Write Skew
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL Serializable	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Reproduced from [1] - “Table 4. Isolation Types Characterized by Possible Anomalies Allowed.”

As discussed in Section 1.2.1, the ACID semantics of eXist-db whilst acceptable for the projects that suit it, are much weaker than we require for FusionDB. eXist-db provides no user controllable transactions, internally it provides a `Txn` object that is often required when writing to the database, however this object in reality just manages transaction demarcation for its database log. In eXist-db, writes are immediate, and visible to all other transactions, likewise aborting a `Txn` does not cause previous statements to be rolled-back. The commit and abort mechanisms of `Txn` strictly exist for the purposes of attempting to return the database to a consistent state during crash recovery by replaying the database log.

- Atomicity

  With eXist-db, the atomicity is provided internally though multi-reader/single-writer locks for Documents, and exclusive locks for everything else including Collections and B+ tree page files.

- Consistency

  Unfortunately, as there are no true transactions in eXist-db, there is no real mechanism for taking the database from one consistent state to the next.

- Isolation

  The effective Isolation level in eXist-db is approximately *Read Uncommitted*.

- Durability

  eXist-db does take serious strides to ensure durability. It provides both a WAL which is *fsynced* to disk, and a synchronization task which periodically flushes dirty in-memory pages to persistent disk.

Fortunately for us, RocksDB provides the Durability and Atomicity properties of ACID. For durability, data is written to both in-memory and a WAL upon commit, in-memory updates are later batched and flushed to disk. Through switching to RocksDB for its storage engine, FusionDB also utilises RocksDB's WAL instead

of eXist-db's, thus discarding eXist-db's previously error prone crash recovery behaviour. This results in greater confidence of recovery from those system crashes that are beyond the control of our software. For Atomicity, RocksDB provides *Write Batches* where batches of updates may be staged and then applied atomically, in this manner all updates succeed together or fail. However, RocksDB leaves us to build the level of Isolation and Consistency that we wish to gain ourselves.

2.2.1. Transactions for FusionDB

Utilising a number of features provided by RocksDB we were able to build a level of isolation for FusionDB which is at least as strong as *Snapshot Isolation*. Firstly, we repurposed eXist-db's `TransactionManager` and associated `Txn` object to provide real user controllable database transactions. Secondly, due to its previous nature, whilst the `Txn` object was often required for write operations in eXist-db, it was rarely required for read operations, this meant modifying a great deal of eXist-db's internal APIs so that a `Txn` is always required when reading or writing the database.

Internally in FusionDB, when a transaction is begun, we make use of RocksDB's MVCC capability and create a Snapshot of the database. These Snapshots are cheap to create, and scale well up to hundreds of thousands, at which point they may significantly slow-down flushes and compactions. However, we think it unlikely that we would need to support hundreds of thousands of concurrent transactions on a single instance. Each snapshot provides an immutable point-in-time view of the database. A reference to the snapshot is placed into the `Txn` object, and is used for every read and scan operation upon the database by that transaction. When only considering reads, this by itself is enough to provide `Snapshot Isolation`, we can then, in fact, remove all of the database read locks that eXist-db used. As the Snapshot is immutable, we no longer need the read locks as no concurrent transaction can modify it.

To provide the isolation for write operations, when a transaction is begun we also create a new Write Batch and hold a reference to it in the `Txn`. This Write Batch sits above the Snapshot, and in actuality all reads and writes of the transaction go to the Write Batch. When the transaction writes to the database, it is actually writing to the Write Batch. The Write Batch stages all of the writes in order in memory, the database is not modified at this point. When the transaction reads from the database, it actually reads from the Write Batch. The Write Batch first attempts to answer the read for a key from any staged updates, if there is no update staged for the key, it falls through to reading from the Snapshot. As each transaction has its own Write Batch, which is entirely in memory, any writes made before committing the transaction are only visible within the same transaction. At commit time, the Write Batch of the (transaction) (`Txn`) is written atomi-

cally to the database, or if the transaction is aborted the memory is simply discarded, as no changes were made there are no changes to roll-back/undo. Whether committed or aborted, when the transaction is complete we release the database snapshot. The semantics are stronger than *Read Uncommitted* because no other transaction can read another transaction's Write Batch, stronger than both *Read Committed* and *Repeatable Read* because every read in the transaction is repeatable due to the snapshot, but also different than *Repeatable Read* as it could exhibit *write skew*. Like reads, for writes, we find that combining a Write Batch with a Snapshot enables us to maintain *Snapshot Isolation* semantics. In FusionDB, we could remove the Write Locks taken by eXist-db upon the database, but we have not yet done so, the write locks are only held for short durations rather than the duration of the transaction, however we are considering providing a configuration option which would extend them to the transaction lifetime, thus yielding *Serializable* isolation.

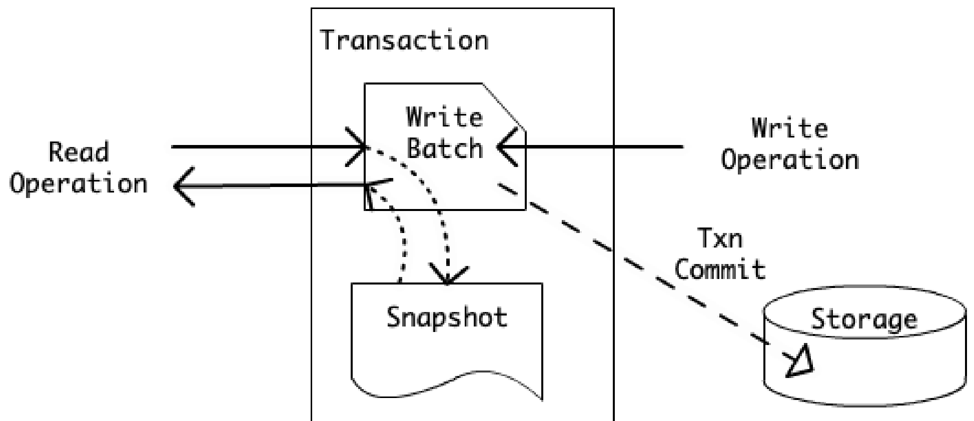


Figure 1. FusionDB Transaction Architecture

2.2.2. FusionDB Transactions and XQuery

At present FusionDB reuses eXist-db's XQuery engine but enforces that an XQuery is executed as a single transaction. This means that an XQuery will either commit all updates atomically or abort, this is in contrast to eXist-db, where there is no atomicity for an XQuery. Consider the XQuery listed in Figure 2, when this query is run in eXist-db if an error occurs during the `update insert` statement, the `/db/test.xml` document will have already been inserted into the database but will be missing its update, and so will have an incorrect balance! In FusionDB because a transaction maintains isolation and commits or aborts atomically, if an error occurs anywhere within the query, the document would never be inserted into the database.

```
1
2 import module namespace xmldb = "http://exist-db.org/xquery/xmldb";
```

```
  3
  4 let $uri := xmldb:store(
  5         "/db",
  6         "some-account.xml,
  7         <account currency="gbp" id="223344"><balance>0</balance></
account>)
  8 return
  9     update insert <balance>9.99</data> into doc($uri)/test
 10
```

Figure 2. Simple Compound Update XQuery

Whilst an XQuery does execute as a single transaction, FusionDB also provides a mechanism akin to *sub-transactions*. Sub-transactions are exposed to the XQuery developer naturally via XQuery *try/catch* expressions. The body of each try clause is executed as a sub-transaction, i.e. an atomic unit, if any expression within the *try body* raises an error, then all expressions within the try body are atomically aborted and the catch clause is invoked, otherwise all expressions in the try body will have been correctly executed. These sub-transactions permit the XQuery developer a fine level of control over their query. Consider the XQuery listed in Figure 3, with FusionDB if an error happens with any of the operations inside try body clause, the sub-transaction is atomically aborted, and so no documents are moved, the task log file then records the failure. With eXist-db due to a lack of atomicity, it an error occurs whilst moving one of the documents, it is entirely possible that some documents could have already been moved even though the query recorded the failure in the task log, thus meaning that the database is no longer logically consistent.

```
  1
  2 import module namespace xmldb = "http://exist-db.org/xquery/xmldb";
  3
  4 let $archived-uris :=
  5     try {
  6         for $record in collection("/db")/record[date lt
xs:date("2001-01-01")]
  7         let $uri := document-uri(root($record))
  8         let $filename := replace($uri, ".*/(.+)", "$1")
  9         return
 10             (
 11                 xmldb:move("/db", "/db/archive", $filename),
 12                 update insert
 13                         <entry>Archived {$uri}</entry>
 14                             into doc("/db/archive-log.xml")/log
 15             )
```

```
16     } catch * {
17         <archive-failure>{$err:code}</archive-failure>
18     }
19 return
20     xmldb:store(
21             "/db",
22             "task-log-" || current-dateTime() || ".xml",
23             <task id="123">{$archived-uris}</task>)
24
```

Figure 3. XQuery with Try/Catch Sub-transaction

XQuery allows try/catch expressions to be nested within the try or catch clause of another try/catch expression, likewise FusionDB supports nesting sub-transactions within sub-transactions. By using the standard try/catch recovery facilities of XQuery, we believe that FusionDB naturally does what the user expects with regards to transaction boundaries.

2.2.3. FusionDB Transactions and APIs

FusionDB strives to maintain API compatibility with eXist-db, and as such FusionDB provides the following eXist-db compatible APIs: REST, RESTXQ, WebDAV, XML-RPC and XML:DB. Apart from the XML:DB API, none of the other eXist-db APIs expose any mechanisms for controlling transactions, and regardless eXist-db does not implement the XML:DB API Transaction Service. FusionDB treats each call to any of these APIs as a distinct transaction, to maintain compatibility with eXist-db there are no mechanisms to establish a transaction across multiple API calls. When executing XQuery via these APIs, the use of transactions and sub-transactions as described in Section 2.2.2 apply. In future, it is likely that FusionDB will provide new APIs to allow external applications greater transactional control.

2.3. Concurrency and Locking

As we previously identified, eXist-db had a number of classes of concurrency problems. We needed to study and understand each of these to ensure that they would not also become issues for the components of eXist-db inherited by FusionDB. As part of our company philosophy of giving back to the larger Open Source community, we recognised that we should fix these problems at their source. As such, we undertook a code-audit project whereby we identified and fixed many concurrency and locking issues directly in eXist-db, we subsequently backported these fixes to the fork of eXist-db that we use for FusionDB. This backporting also enabled us to further increase our confidence in our changes by showing that not only did our code pass the new tests that we had created, but

that the modified version of eXist-db could pass the existing full-test suites of both eXist-db and FusionDB.

The changes we made to the locking and concurrency mechanisms in eXist-db were numerous and far-reaching. We have recently published two comprehensive technical reports detailing the problems and the solutions that we implemented [43] [44]. These comprehensive technical improvements have been incorporated into both eXist-db 5.0.0 and FusionDB. Instead of reproducing those technical reports within this paper, we will briefly highlight the improvements that were made and their impact for FusionDB.

2.3.1. Incorrect Locking

We identified and addressed many issues in the eXist-db codebase that were the result of incorrect locking, these included:

- Inconsistent Locking, whereby locking was applied differently to the same types of objects at varying places throughout the codebase. We documented the correct locking pattern, and made sure that it was applied consistently. This gave us improved deadlock avoidance of different types of locks which must interleave, e.g. Collection and Document locks, which are now always acquired and released in the same order.
- Insufficient/Overzealous Locking, whereby in some code paths objects were accessed either without locks or with more locking than is required for a particular operation. We modified many code paths to ensure that the correct amount of locking is used.
- Incorrect Lock Modes, where shared reader/writer locks were used, we repaired some cases whereby the wrong lock mode was used. For example, where a read lock was taken but a write was performed, or vice-versa.
- Lock Leaks and Accidental Release, whereby a lock is never released, is released too soon, or too often for reentrant locks. We introduced the concept of Managed Locks, and deployed them throughout the codebase. Our Managed Locks make use of Java's *try-with-resources* expression, to ensure that they are always released, this is done automatically and in line with the developer's expectations.

2.3.2. Lock Implementations

We identified that alongside standard Java Lock implementations, eXist-db also made use of two proprietary lock implementations. Whilst no issues with the lock implementations had been directly reported, we questioned the likely correctness of them, theorising that they could be a contributor to other reported problems with database corruption. As part of our need to understand them we were able to research and develop an incomplete provenance for them.

1. Collection Locks. eXist-db's own `ReentrantReadWriteLock` class was used for its Collection Locks. It was originally copied from Doug Lea's `ReentrantLock`, which was itself superseded by J2SE 5.0's locks. The eXist-db version has received several modifications which make it appear like a multi-reader/single-writer lock, and its naming is misleading, as in actuality it is still a mutually exclusive lock.

   Document Locks. eXist-db's own `MultiReadReentrantLock` class was used for its Document Locks. It was likely copied from the Apache Turbine JCS project which is now defunct. Strangely, this is a multi-reader/single-writer lock, which also appears to support lock upgrading. However, lock upgrading is a well-known anti-pattern which is typically prohibited by lock implementations. The eXist-db version has received several changes which were only simply described as “bug-fixes”.

Ultimately, we felt that the custom Lock Implementations in use by eXist-db were of questionable pedigree and correctness. We replaced them with implementations that we believe to be both correct and reliable. We switched Document Locks to Java's standard `ReentrantReadWriteLock`. Whilst for Collection Locks we switched to `MultiLock` [45] from Imperial College London, which is itself based on Java's standard locking primitives. MultiLock is a multi-state intentioned multi-reader/single-writer lock, thus allowing for concurrent operations on Collection objects.

2.3.3. Asymmetrical Locking

Previously, the correct locking pattern in eXist-db when performing read operations on Documents within a Collection, was to lock the Collection for read access, retrieve the Document(s) from the Collection, lock the Documents for read access, perform the operations on the documents, release the Document locks and then finally release the Collection locks.

We were able to optimise this pattern to reduce the duration that Collection Locks are held for. Our asymmetrical pattern allows the Collection lock to be released earlier, after all the Document locks have been acquired, thus reducing contention and improving concurrent throughput.

2.3.4. Hierarchical Locking

Locks in eXist-db were previously in a flat space with one Lock per-Collection or Document object. Unfortunately, this meant that user-submitted concurrently executing queries could acquire locks for write access in differing orders which would could cause a deadlock between concurrent threads. A deadlock in eXist-db is unresolvable without restarting the system, at which point the Recovery Manager has to attempt to bring the database back to a logically consistent state.

We created a modified version of Gray's hierarchical locking scheme [46] for Collection Locks, whereby the path components of eXist-db's Collection URIs represent the hierarchy. The hand-over locking through this hierarchy that we implemented, coupled with intention locks provided by MultiLock, permit multi-reader/single-writer Collection access, but make it impossible to deadlock between concurrent Collection lock operations. We also added a configuration option (disabled by default) that allows for multi-reader/multi-writer locking of Collections, but its use requires careful application design by the XQuery developer.

In eXist-db, we have not as yet extended the hierarchical locking scheme to cover Documents, and so it is still possible to deadlock between Collection and Document access, if user-submitted queries have different lock schedule ordering for the same resources.

In FusionDB, less locking is required due to our MVCC based snapshot isolation, however at present such deadlocks are still possible for write operations. However, it is possible to resolve a deadlock by aborting a transaction in FusionDB, leaving the database in a consistent and functioning state.

2.3.5. Concurrent Collection Caching

eXist-db made use of a global Collection Cache to reduce both object creation overhead and disk I/O. Unfortunately, this cache was a point of contention for concurrent operation, as accessing it required acquiring a global mutually exclusive lock over the cache. For eXist-db we replaced this Collection Cache with a concurrent data structure called Caffeine which allows fine-grained concurrent access without explicit locking. For FusionDB, we need such global shared structures to be version aware so that they integrate with our MVCC model. We are working on replacing this shared cache with something similar to Caffeine but also supports MVCC.

2.3.6. New Locking Features

Alongside many technical fixes that we have made to eXist-db, we have also added three new substantial features:

1. A centralised Lock Manager, whereby all locking actions are defined consistently in a single class, and regardless of the underlying lock implementation they present the same API.

2. A Lock Table which is fed by the Lock Manager, and allows the state of all locks in the database system to be observed in real-time. It also provides facilities for tracing and debugging lock leases, and makes its state available via JMX for integration with 3rd-party monitoring systems.

3. A set of annotations named `EnsureLocked` which can be added to methods in the code base. These annotations form a contract which describes the locks that should be held when the method is invoked. When these annotations are enabled for debugging purposes, they can consult the lock table and eagerly report on violations of the locking contracts.

Example 1. Example Use of Locking Annotations

```
private Collection doCopyCollection(final Txn transaction,
    final DocumentTrigger documentTrigger,
    @EnsureLocked(mode=LockMode.READ_LOCK) final Collection
sourceCollection,
    @EnsureLocked(mode=LockMode.WRITE_LOCK) final Collection
destinationParentCollection,
    @EnsureLocked(mode=LockMode.WRITE_LOCK, type=LockType.COLLECTION)
final XmldbURI destinationCollectionUri,
    final boolean copyCollectionMode,
    final PreserveType preserve) {
```

2.4. UUIDs

In FusionDB every Collection and Document is assigned an immutable and persistent UUID (Universally Unique Identifier). Each UUID is a 128-bit identifier, adhering to the *UUID Version 1* specification as defined by IETF (Internet Engineering Task Force) RFC (Request For Comments) 4122. Instead of using the hosts MAC address, instead as permitted by the UUID specification, we use a random multicast address, which we generate and persist the first time a database host is started. This allows us per-host identification for the UUIDs within a multi-host system, but without leaking information about any hosts network settings.

These UUIDs allow the user to refer to a Collection or Document across systems, and retrieve it by its identifier regardless of its location with the database. When a Collection or Document is moved within the database, its UUID remains unchanged, whilst a copy of a Collection or Document will be allocated a new UUID. UUIDs are also preserved across backup and restore. When restoring a backup, the backup will notify the user of any conflicts between Collections and Documents in the database and the backup that have the same UUID but different database locations.

2.5. Key/Value Metadata

FusionDB offers a key/value metadata store for use with Collections and Documents. Any Collection or Document may have arbitrary metadata in the form of

key/value pairs which are transparently stored alongside it. FusionDB also provides range indexing and search for Collections and Documents based on both the keys and values of their associated Metadata. For example, the user can formulate queries like, "Return me all of the documents which have the metadata keys `town` and `country`, with the respective metadata values, `Wiggaton` and `United Kingdom`. Within FusionDB updates across are atomic and consistent across data-models, so for example, it is impossible for a Document and the Key/Value Metadata associated with that document to be inconsistent with respect to each other even under concurrent updates. Although not yet exposed via XQuery, internally there are also iterator based APIs for efficiently scanning over Collections and Documents metadata.

At present our Key/Value store, is just one of the small ways in which we expose the multi-model potential of FusionDB.

2.6. Online Backup

For backing up a FusionDB database we are able to make use of RocksDB's MVCC facilities to create snapshots and checkpoints. We provide two mechanisms for backing up the database, 1) a Full Document Export, and 2) a Checkpoint Backup. The Checkpoint Backup process is both light-weight and fast, and is the preferred backup mechanism for FusionDB. The Full Document Export process exists only for when the user needs a dump of the database for use with 3rd party systems such as eXist-db.

2.6.1. Full Document Export

The Full Document Export mechanism will be familiar to eXist-db users as it's very similar to the primary backup format for eXist-db, and is in fact compatible. The export results in a directory or zip file, which contains a complete copy of every document from the Collections that the user chose to backup. The output directory or zip file is also structured with directories representing each sub-Collection that is exported. Each directory in the output contains a metadata file named `__contents__.xml` which contains the database metadata for the Collection and its Documents.

The metadata files have been extended from eXist-db for FusionDB to also contain the key/value metadata (see Section 2.5) that a user may have associated with Collection or Documents, and the UUID (see Section 2.4) of each Collection and Document. We should also explicitly state that this Backup is transactional, and as transactions in FusionDB are snapshots of the database, the export has the advantage of being point-in-time consistent unlike in eXist-db. In addition, due to removing read-locks in FusionDB thanks to our snapshot isolation, unlike eXist-db the backup process will not block access for reads or writes to the database.

2.6.2. Checkpoint Backup

Checkpoint backups are provided by the underlying RocksDB storage engine and are near-instantaneous. They exploit both the MVCC and append-only nature of the storage engine's database files. When a checkpoint is initiated, a destination directory is created and if the destination is on the same filesystem then the files making up the current state of the live database will be hard-linked into it, otherwise they are copied over to it. Once the checkpoint has completed, a backup process operates directly from the checkpoint to provide a version of the database files that are suitable for archiving. The first part of this process is relatively simple, and really provides just the RocksDB database files holding the data for the backup and some metadata describing the backup. The second part of the backup process makes sure to copy any binary documents present in the database to the backup directory as well, to do this it starts a second read-only database process from the checkpoint, scans the binary documents in the database, making of copy of the filesystem blob of each one. When initiating a Checkpoint Backup, the user can choose either a full or incremental backup. As the backup process operates from a checkpoint, it cannot block concurrent operations accessing the database. The file format of the Checkpoint Backup is proprietary to FusionDB and should be treated as a black-box.

2.7. BLOB Store

Similarly to our work on Locking (see Section 2.3), we recognised that there were a number of problems with the handling of Binary Documents (i.e. non-XML documents), that we had inherited with eXist-db. We again decided to address these by contributing back to the wider Open Source community, as such we developed a new BLOB (Binary Large Object) store component[3] for use within eXist-db. This new BLOB Store is available in FusionDB, and a Pull Request that we prepared will likely be merged into eXist-db in the near future.

Although we have recently reported on the technical detail of the new BLOB Store [47], it is relevant to highlight two key points of its design:

- Lock Free. The entire Blob Store operates without any locks whatsoever, instead it operates as an atomic state machine by exploiting CAS (Compare-And-Swap) CPU instructions. This design decision was taken to try and increase concurrent performance for Binary Document operations.

  Deduplication. The Blob Store only stores each unique Binary Document once. When a copy of a Document is made, a reference counter is incremented, and conversely when a copy is deleted the reference counter is decremen-

[3]RocksDB has recently developed an extension called BlobDB, which is still highly experimental. Like our BLOB Store it also stores Binary Document file content in a series of files on the filesystem, but as far as we are aware does not yet offer deduplication.

ted. The Binary Document's content is only deleted once the reference count reaches zero. This design decision was made to try and reduce disk IO in systems which make use of many Binary Documents.

Importantly with regards to FusionDB, the new BLOB Store was designed not to reuse eXist-db's B+Tree storage engine, but instead persists a simple Hash Table. In FusionDB we have also replaced the BLOB Store's persistent Hash Table with a RocksDB Column Family, which means that updates to Binary Document objects are also transactionally consistent.

3. High-level Architecture

In this section, we detail several of the high-level architectural aspects of FusionDB.

3.1. Programming Language

FusionDB is written in a combination of both Java 8 and C++14 and is available for Windows, macOS, and Linux on x86, x86_64, and PPC64LE CPUs. This came about predominantly because eXist-db 5.0.0 is written in Java 8, whilst RocksDB is written in C++14 and RocksJava (the RocksDB Java API) is written in Java 7. Evolved Binary have been contributors to the RocksJava API since the adoption of RocksDB for FusionDB's storage engine. The RocksJava API is comprehensive, but typically lags behind the RocksDB C++ API, and so Evolved Binary have make many Open Source contributions to add missing features and fix issues.

The RocksJava API makes heavy use of Java's JNI (Java Native Interface) version 1.2 for calling C++ methods in RocksDB. Unfortunately when calling C++ code from Java (or vice-versa) via JNI, there is a performance penalty each time the Java/C++ boundary is traversed. After benchmarking different call mechanisms across the JNI boundary [48], we were able to improve the situation somewhat. However, when callbacks from C++ to Java involve millions of calls, perhaps because of iterating over large datasets, the cost of the JNI boundary accumulates quickly and has a large impact on performance. To avoid such a penalty and ensure good performance, several components of FusionDB which communicate directly and frequently with RocksDB were rewritten in C++. These components are then compiled directly into a custom RocksDB library that is used by FusionDB.

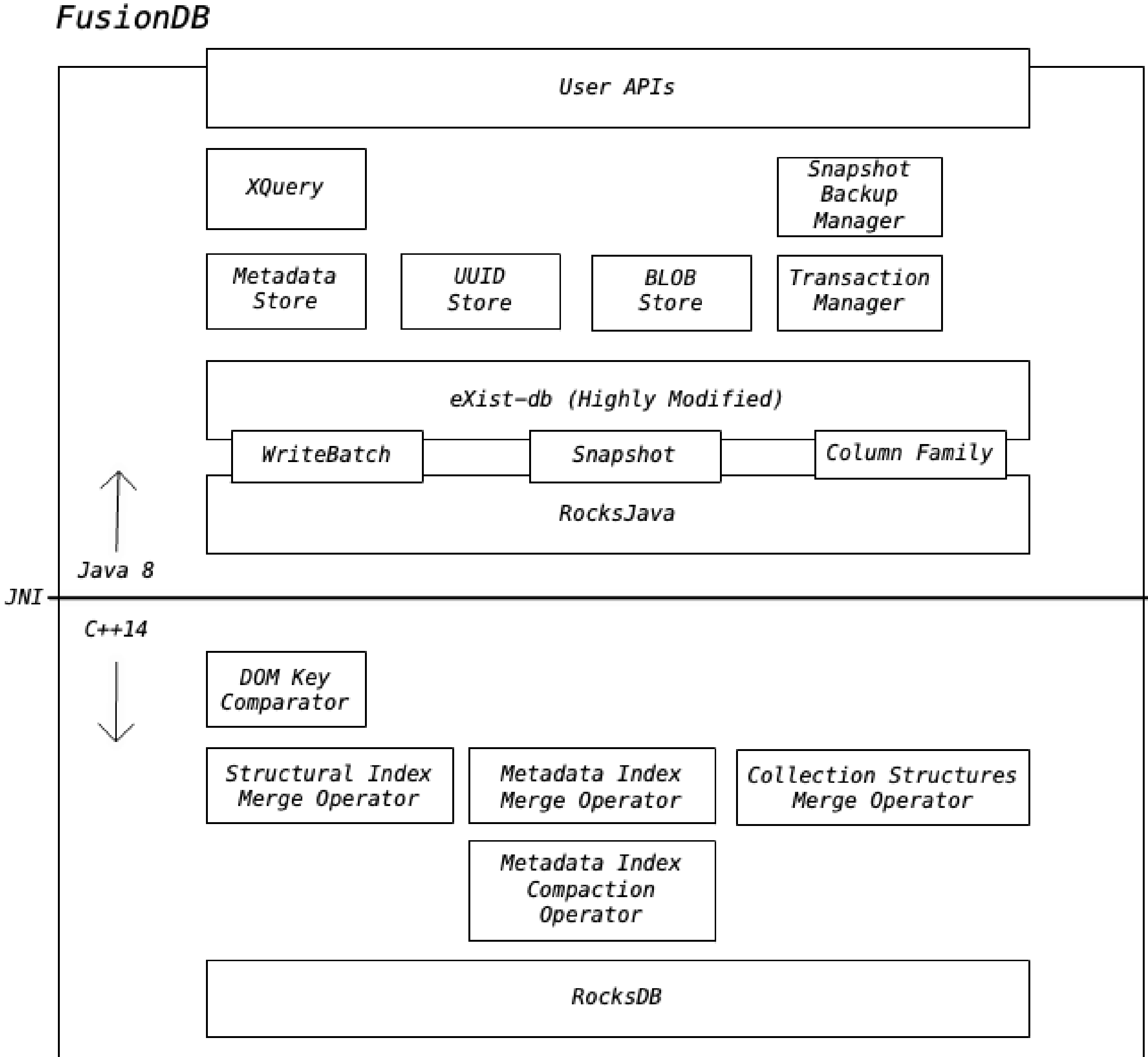


Figure 4. FusionDB - Components by Language

3.2. Column Families

As discussed in Section 2.1.3, RocksDB provides a feature called *Column Families*. These Column Families allow you to separate the key/value pairs that make up the database into arbitrary groupings. By storing homogeneous groups of keys and values into distinct Column Families, we can configure each Column Family in a manner that best reflects the format of those keys and values, and the likely access patterns to them. Modifications across Column Families remain atomic.

When we replaced eXist-db's B+Tree based storage engine with RocksDB, we carefully ported each distinct sub-system in eXist-db that used a B+Tree to one or more RocksDB Column Families. We also replaced eXist-db's non-B+Tree based Symbol Table and its indexes with a newly designed highly concurrent Symbol

Table which is persisted to several Column Families. In addition the new sub-systems that we added for Key/Value Metadata (see Section 2.5) and UUID locators (see Section 2.4) also make use of Column Families for storing data and indexes.

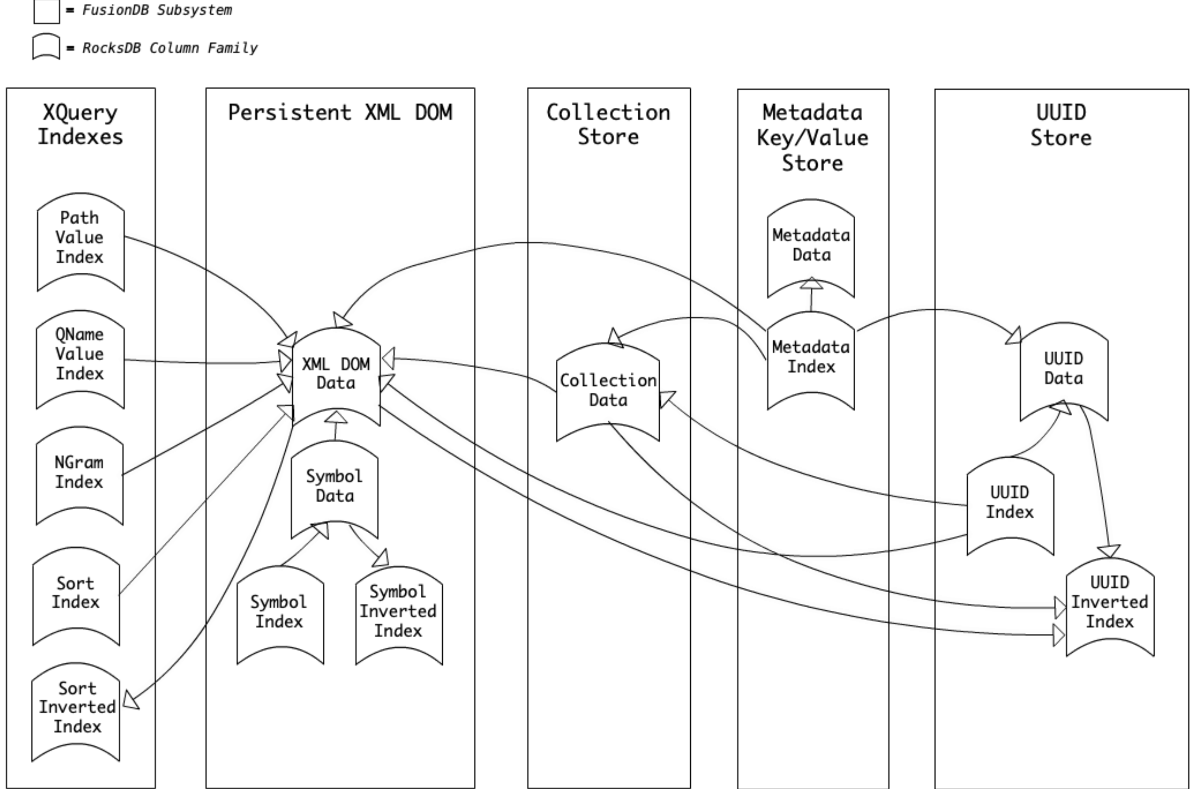


Figure 5. Main Column Families used by FusionDB

4. Conclusion

In 2014, the inception of this project came about as a response to a number of issues that we had identified over a period of several years from both users of Open Source NXDs and developers. We therefore set out to build a better modern Open Source database system which could handle XML Documents as well as other data models.

Initially we opted to fork and modify eXist-db as the base for FusionDB. Our plan was to firstly, fork eXist-db and replace its storage engine with RocksDB, before undertaking further radical changes to our fork of eXist-db. Then secondly, to build out new features, both alongside and with our fork of eXist-db. At that time, we believed that forking the eXist-db codebase with which we were familiar, would result in a faster route to delivery for FusionDB. Our initial estimate was calculated at between six and twelve months of development time. Unfortunately, this proved not to be the case, in fact quite the opposite! eXist-db has evolved over 18 years, and as such is a feature rich and complex product. As an

Open Source project, many developers have contributed and moved on, and there are several areas that are undocumented, untested, and no longer well understood by its current development team. Our goal, was to pass 100% of eXist-db's test suite with FusionDB. This proved to be a tough although not insurmountable challenge, however, achieving this revealed many bugs and incorrect tests in eXist-db which we also had to fix (and chose to contribute back to the eXist-db Open Source project). Whilst we have gained much by forking eXist-db, with hindsight we believe that it would have been quicker to develop a new eXist-db compatible system from scratch without forking eXist-db.

With FusionDB, in the first instance, we have now addressed many of the issues identified by users and developers. RocksDB has given us much greater stability, crash resilience, recovery, and performance. We have created an ACID transactional database system which offers Snapshot Isolation and ensures consistency and atomicity. We have invested a great deal of time and effort in to improving the locking situation of eXist-db, which when coupled with our Snapshot Isolation has also reduced the number and duration of locks that we require in FusionDB. Reduced locking has improved performance generally and decreased contention, thus also improving vertical scalability. Likewise, it has also prevented system maintenance tasks such as Backup or Reindexing from blocking access to the database, as these now have their own dedicated snapshot on which to act. Additionally we have also replaced and contributed new subsystems which add Key/Value Metadata, UUID locators, and improved Binary Document storage. This paper has both described much of the design rationale behind these technical achievements, and revealed the architecture of their implementation.

Whilst we have not yet completed work on the issues of complex graph-like cross-reference queries, or clustering to deliver horizontal scalability, they are firmly on our roadmap. The construction of FusionDB has not been an easy path to navigate, but ultimately we felt that the work was both justified and deserved by the XML community. We believe that FusionDB has a huge amount of potential, and we are excited to both share it with the world and develop it further.

5. Future Work

A great deal of research and development work has gone into the development of FusionDB. Some of this work, such as that around Locking and Binary Storage, has been contributed back to eXist-db, whilst more modest improvements have been contributed back to RocksJava.

We are approaching the stage where we believe we could release a first version of FusionDB, however before that becomes a reality, there is still work to complete in the short-term concerning:

- Collection Caching.

  The Collection Cache of eXist-db is shared between multiple transaction but is yet version aware. The Collection cache was originally designed to reduce Java Object creation and memory-use by ensuring only one Object for each distinct Collection, and reduce disk I/O by avoiding the need to frequently re-read Collections from disk. To cope with transactional updates upon cached in-memory Collection objects, the Collection Cache needs to be either, removed entirely, or revised to be MVCC aware so that a transaction is accessing the correct in-memory version of the on-disk Collection.

- Locking and Transactions.

  Whilst we have hugely improved the current state of locking in eXist-db, for FusionDB there are many more opportunities for reducing the number of locks whilst preserving transaction isolation and consistency. One obvious area of work, would be looking for any advantages in replacing FusionDB's transactions with the new facilities recently available in RocksDB for pessimistic and optimistic transactions.

- Performance.

  Ultimately our goal is to push FusionDB to out-perform any other Document Database. Our first performance goal is to have FusionDB out-perform eXist-db on any operation or query. At present, FusionDB is faster than eXist-db for many operations, but slower than eXist-db for several others. Our changes so far to the fork of eXist-db used by FusionDB have been far-reaching but somewhat conservative, we have focused on carefully reproducing existing behaviour. We are looking forward to drastically improving performance, by exploiting the many untapped optimisations that our new technology stack affords.

- Licensing.

  Our goal has always been to release FusionDB as Open Source software, and this has not changed. However, we want to ensure that we choose an appropriate license or licenses, that enable us to build the best possible community and software. We were previously considering a dual-licensed approach, whereby users could choose AGPLv3 or a Commercially licensed exemption to AGPLv3.

  Recent developments in software licensing require us to properly reevaluate this choice. Concrete examples of this in the database market are:

  - Redis previously licensed its open source version under a BSD 3-clause, and the modules for that under AGPLv3. Its Enterprise edition is only available under a commercial license without source code. Around the August 22nd 2018, Redis relicenced its modules from AGPLv3 to Apache 2.0 modified with Commons Clause, changing them from Open Source to Source Available.

- MongoDB's database was previously licensed under AGPLv3. On October 16th 2018, MongoDB relicensed its software under a new license SSPL (Server Side Public License) it created to solve their perceived problems with AGPL. Whether SSPL is an Open Source or Source Available license is still to be determined by its submission to the OSI.
- Neo4j previously licensed its Community Edition under GPLv3, and its Enterprise edition under AGPLv3. On November 15th 2018, Neo4j changed to an Open Core model, whereby their Enterprise Edition is now only available under a commercial license without source code.

The reason for these licensing changes have often been cited as protecting commercial interests of Open Source companies. However it is clear that each of these companies is taking a different route to achieve a similar outcome. We believe that further study of the outcomes of these changes is merited, with a focus on their acceptance or rejection by their respective previous open source users.

In the medium term, areas of future work that are already under consideration are:

- Further Document Models

  Of primary concern is support for natively storing JSON documents. Several options exist for querying across XML and JSON document models, including XQuery 3.1 Maps/Arrays and JSONiq, further work is needed to identify the best approach for FusionDB. In addition, some research has already been undertaken into also storing Markdown and HTML5 documents natively, and will likely be further expanded upon.
- Distributed Database

  After correctness and single-node performance, we consider this to be the most important feature of modern database that are designed to scale. Further research into establishing a multi-node shared-nothing instance of FusionDB is highly desirable.
- Graph Database

  As discussed (see Section 1.1) many users have complex cross document query requirements that would likely benefit from more complex graph based linking and querying. The key/value facilities of RocksDB have been previously been demonstrated by ArangoDB and DGraph as a feasible base for building Graph database models. Integrating a graph model into FusionDB, and the subsequent opportunities for cross-model querying is an interesting topic for further research in FusionDB.

Bibliography

[1] Hal Berenson. Phil Bernstein. Jim Gray. Jim Melton. Elizabeth O'Neil. Patrick O'Neil. June 1995. *A Critique of ANSI SQL Isolation Levels*. Association for

Computing Machinery, Inc. http://research.microsoft.com/pubs/69541/tr-95-51.pdf .

[2] Wolfgang Meier. eXist-db. 2006-10-13T09:35:53Z. *Re: [Exist-open] Lock (exclusive-lock, etc) does anybody have a book, tutorial, or helpful explanation?*. exist-open mailing list. https://sourceforge.net/p/exist/mailman/message/14675343/ .

[3] Wolfgang Meier. eXist-db. 2005-07-26T07:15:39Z. *Initial checkin of logging&recovery code.*. GitHub. https://github.com/eXist-db/exist/commit/1ed4d47f01c9ee2ede#diff-16915756b76d37e10eba8b939a1e2f40R1648.

[4] Wolfgang Meier. Adam Retter. eXist-db. 2018-04-23T20:21:37+02:00. *[bugfix] Fix failing recovery for overflow DOM nodes (>4k). Addresses #1838*. GitHub. https://github.com/exist-db/exist/commit/6467898.

[5] Pierrick Brihaye. eXist-db. 2007-02-13T16:13:31Z. *Made a broader use of transaction.registerLock(). Left 2 methods with the old design (multiple collections are involved).*. GitHub. https://github.com/eXist-db/exist/commit/cd29fef34f91d471d79966b963d1657fd9186f89.

[6] Dmitriy Shabanov. eXist-db. 2010-12-14T17:44:07Z. *[bugfix] The validateXMLResourceInternal lock document, but do not unlock with a hope that it will happen up somewhere. Link lock with transaction, so document lock will be released after transaction completed or canceled.*. GitHub. https://github.com/eXist-db/exist/commit/5af077cd441039d9b8125a9149f399b9bd8ee95c.

[7] Adam Retter. eXist-db. 2018-10-31T19:12:22+08:00. *[bugfix] Locks do not need to be acquired for the transaction life-time for permission changes*. GitHub. https://github.com/eXist-db/exist/commit/0faee22bc792629d625807319459a164d00691c1.

[8] eXist-db. *eXist-db B+ tree*. GitHub. https://github.com/eXist-db/exist/blob/eXist-4.5.0/src/org/exist/storage/btree/BTree.java#L26.

[9] dbXML. *dbXML 1.0b4 B+Tree*. SourceForge. http://sourceforge.net/projects/dbxml-core/files/OldFiles/dbXML-Core-1.0b4.tar.gz/download.

[10] Christian Grün. BaseX. 2014-05-23T17:54:00Z. *Transaction Management - Concurrency Control*. BaseX. http://docs.basex.org/index.php?title=Transaction_Management&oldid=10646#Concurrency_Control.

[11] J. Chris Anderson, Jan Lehnardt, and Noah Slater. Apache. 2010-02-05. *CouchDB - The Definitive Guide*. *Storing Documents.* 1st. O'Reilly. http://guide.couchdb.org/draft/documents.html.

[12] MongoDB, Inc.. 2018-11-29. *What is MongoDB?*. https://www.mongodb.com/what-is-mongodb.

[13] Marklogic Corporation. 2018-11-29. *MarkLogic Application Developers Guide*. *Working with JSON.* https://docs.marklogic.com/guide/app-dev/json.

[14] Marklogic Corporation. 2018-11-29. *MarkLogic Application Developers Guide*. *Understanding Transactions in MarkLogic Server.* https://docs.marklogic.com/guide/app-dev/transactions.

[15] David Gorbet. Marklogic Corporation. 2018-11-30. *I is for Isolation, That's Good Enough for Me! - MarkLogic*. https://www.marklogic.com/blog/isolation/.

[16] Sinclair Target. Two-Bit History. 2017-09-21. *The Rise and Rise of JSON*. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html.

[17] Christian Grün. BaseX. 2014-11-20T14:02:00Z. *SQL Module*. BaseX. http://docs.basex.org/index.php?title=SQL_Module&oldid=11101.

[18] Dan McCreary. XQuery Examples Collection Wikibook. 2011-04-14T16:42:00Z. *XQuery SQL Module*. Wikibooks. https://en.wikibooks.org/wiki/XQuery/XQuery_SQL_Module.

[19] Oracle. 2015-07-10. *Introduction to Berkeley DB XML*. *Database Features.* Oracle. https://docs.oracle.com/cd/E17276_01/html/intro_xml/dbfeatures.html.

[20] Pete Aven. Diane Burley. MarkLogic. 2017-05-11. *Building on Multi-Model Databases*. *How to Manage Multiple Schemas Using a Single Platform.* 1st. 24-26. O'Reilly. http://info.marklogic.com/rs/371-XVQ-609/images/building-on-multi-model-databases.pdf.

[21] Philip Lehman. S BING YAO. 1981. *Efficient Locking for Concurrent Operations on B-Trees*. ACM. *ACM Transactions on Database Systems*. 6. 4. 650-670.

[22] Anastasia Braginsky. Erez Petrank. Technion - Israel Institute of Technology. 2012. *A Lock-Free B+tree*. *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. *SPAA '12*. ACM. 58-67. 978-1-4503-1213-4. 10.1145/2312005.2312016.

[23] Justin Levandoski. David Lomet. Sudipta Sengupta. Microsoft Research. 2014-04-08. *The Bw-Tree: A B-tree for new hardware platforms*. IEEE. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 978-1-4673-4910-9. 10.1109/ICDE.2013.6544834.

[24] Lars Arge. 1995. *The Buffer Tree: A New Technique for Optimal I/O Algorithms*. BRICS, Department of Computer Science, University of Aarhus. *Lecture Notes in Computer Science*. 995. *WADS 1995*. Springer.

[25] Gerth Stølting Brodal. Rolf Fagerberg. 2003. *Lower Bounds for External Memory Dictionaries*. BRICS, Department of Computer Science, University of Aarhus. Society for Industrial and Applied Mathematics. *Proceedings of the Fourteenth*

Annual ACM-SIAM Symposium on Discrete Algorithms. *SODA '03*. 546-554. 0-89871-538-5.

[26] brandur. 2017-05-07. *The long road to Mongo's durability*. https://brandur.org/fragments/mongo-durability.

[27] Nassyam Basha. 2018-04-01. *Database Lost write and corruption detections made easy with dbcomp - 12.2*. Oracle User Group Community. https://community.oracle.com/docs/DOC-1023009.

[28] Robert Newson. 2017-01-19. *[COUCHDB-3274] eof in couch_file can be incorrect after error - ASF JIRA*. Apache Software Foundation. https://issues.apache.org/jira/browse/COUCHDB-3274.

[29] Jeffrey Aguilera. 2005-10-07T08:08:00Z. *[DERBY-606] SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE fails on (very) large tables - ASF JIRA*. Apache Software Foundation. https://issues.apache.org/jira/browse/DERBY-606.

[30] *Issues · jankotek/mapdb*. 2019-01-20. GitHub. https://github.com/jankotek/mapdb/issues?utf8=%E2%9C%93&q=is%3Aissue+corrupt.

[31] Ryan S. Dancey. *License-discuss Mailing List*. *OpenLDAP license.* 2001-04-09T23:22:48Z. http://lists.opensource.org/pipermail/license-discuss_lists.opensource.org/2001-April/003156.html.

[32] Jung-Sang Ahn. Chiyoung Seo. Ravi Mayuram. Rahim Yaseen. Jin-Soo Kim. Seung Ryoul Maeng. 2016-03-01. 2015-05-20. *ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys*. IEEE. *Published in: IEEE Transactions on Computers*. 65. 3. 902-915. 10.1109/TC.2015.2435779.

[33] Patrick O'Neil. Edward Cheng. Dieter Gawlick. Elizabeth O'Neil. 1996. *The Log-structured Merge-tree (LSM-tree)*. *Acta Informatica*. 33. Springer-Verlag New York, Inc.. 351-385. 10.1007/s002360050048.

[34] Ilya Grigorik. 2012-02-06. *SSTable and Log Structured Storage: LevelDB*. https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/.

[35] Siying Dong. 2015-02-27. *WriteBatchWithIndex: Utility for Implementing Read-Your-Own-Writes*. https://rocksdb.org/blog/2015/02/27/write-batch-with-index.html.

[36] Dhruba Borthakur. Facebook. 2013. *The Story of RocksDB*. *Embedded Key-Value Store for Flash and RAM.* https://github.com/facebook/rocksdb/blob/gh-pages-old/intro.pdf?raw=true.

[37] Symas Corporation. 2019. *LMDB TECHNICAL INFORMATION*. *Other Projects.* https://symas.com/lmdb/technical/#projects.

[38] Facebook. GitHub. 2019. *rocksdb/USERS.md at v5.17.2 · facebook/rocksdb*. *Users of RocksDB and their use cases. Other Projects.* https://github.com/facebook/rocksdb/blob/v5.17.2/USERS.md.

[39] Facebook. GitHub. 2019. *rocksdb/block_builder.cc at v5.17.2 · facebook/rocksdb*. https://github.com/facebook/rocksdb/blob/v5.17.2/table/block_builder.cc#L10.

[40] Facebook. GitHub. 2019. *Prefix Seek API Changes · facebook/rocksdb Wiki*. https://github.com/facebook/rocksdb/wiki/Prefix-Seek-API-Changes.

[41] Facebook. GitHub. 2019-01-23. 2019-01-21. *RocksDB - Wikipedia*. *Integration.* https://en.wikipedia.org/wiki/RocksDB#Integration.

[42] Dan Ports. Kevin Grittner. 2012-08. *Serializable Snapshot Isolation in PostgreSQL*. VLDB Endowment. *Proc. VLDB Endow.*. 5. August 2012. 1850-1861. 10.14778/2367502.2367523.

[43] Adam Retter. *Locking and Cache Improvements for eXist-db*. 2018-02-05. https://www.evolvedbinary.com/technical-reports/exist-db/locking-and-cache-improvements/locking-and-cache-improvements-20180205.pdf.

[44] Adam Retter. *Asymmetrical Locking for eXist-db*. 2018-02-05. https://www.evolvedbinary.com/technical-reports/exist-db/asymmetrical-locking/asymmetrical-locking-20180205.pdf.

[45] Gudka Khilan. Susan Eisenbach. *Fast Multi-Level Locks for Java*. *A Preliminary Performance Evaluation.* 2010. *EC^2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly*. https://www.cl.cam.ac.uk/~kg365/pubs/ec2-fastlocks.pdf.

[46] Gudka Khilan. 1975. *Granularity of Locks in a Shared Data Base*. *Proceedings of the 1st International Conference on Very Large Data Bases*. VLDB '75. 10.1145/1282480.1282513. 978-1-4503-3920-9. ACM. 428-451.

[47] Adam Retter. *BLOB Deduplication in eXist-db*. 2018-11-27. https://blog.adamretter.org.uk/blob-deduplication/.

[48] Adam Retter. *JNI Construction Benchmark*. *Results.* 2016-01-18. https://github.com/adamretter/jni-construction-benchmark#results.

xqerl_db: Database Layer in xqerl

Zachary N. Dean
<contact@zadean.com>

Abstract

xqerl, an open-source XQuery 3.1 processor written in Erlang, now has an internal database layer for persistent storage of XML and JSON documents as well as other unparsed resources. This paper will discuss overall layout of the data layer, some internal workings of databases, and work that is still to be done concerning document querying and indexing.

Keywords: XML Database, XQuery, Erlang, Big Data

1. Introduction

There were many reasons for adding a persistent data layer to xqerl. Parsing of data, be it XML or JSON, can be one the most resource-consuming parts of the system. [1] To limit this cost and time, saving the parsed and decomposed data means parsing only happens once on input. To fully implement the Update Facility for XQuery without altering files on the filesystem, or only being able to handle in-memory changes of the copy-modify statement, a persisted store was also needed.

Also, to be useful in any number of use-cases, be it applications with high insertion transaction rates, or with the need to store huge amounts of data, the data backend should have several qualities. The data storage should be relatively compact when compared to the unparsed data that it contains. There should also be as few limitations as possible on the size and complexity of the data that is input. Nodes should be able to be quickly accessed regardless of physical location. Data should also be able to be rapidly queried and reconstructed on demand.

This new datastore is an initial attempt to address many of these requirements and is described here.

2. Overview of the System

The entire system can be envisioned as a collection of loosely coupled independent databases with one process keeping track of all databases and their current states. The basic outline of the system is shown in Figure 1. This layout allows each database to handle portions of queries on their own and in parallel.

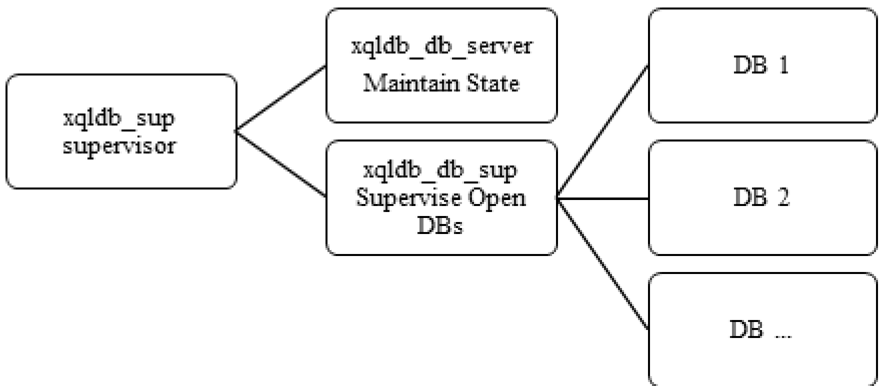


Figure 1. Process Overview - Supervision Tree

The xqldb_sup process is the root of the supervision tree and simply ensures that the entire system stays running. Should either of its child processes, or both, crash, it restarts both anew. The xqldb_db_sup process supervises each open database. If any database should crash, it is restarted singly. The xqldb_db_server process maintains a record of all existing databases and their current status. This process also handles all requests for starting and stopping databases, as well as database discovery in the system.

Each individual database is given a unique 32-bit integer identifier upon creation. Using this method does limit the total number of possible databases the system can address, but it would require creating one database per second for well over 100 years to reach this limit and therefore is imagined to be large enough. The integer identifier is used to create a physical folder on the local filesystem to hold all the databases data files. To avoid a possibly extremely wide directory structure each identifier is encoded to its hexadecimal string value and split into four values of two characters each. This causes each directory to have only a maximum of 256 immediate subdirectories.

Each database represents a node in the URI hierarchy of all document URIs in the system. That is, for each "directory" that contains a document in the system, there exists one database. Databases are dynamically created at the time of the first document insertion into that directory. Collections of documents can be reference either by their base directory or an ancestor directory.

3. Databases

The term "database" as used in this paper is meant to mean a single self-contained, independent group of processes and linked data. It can be considered roughly like a table in a relational database including all its indexes, data and data-files thereof.

Each database consists of one supervisor process and several child processes that each encapsulate a portion of the database and that portion's individual logic. The supervisor process only ensures that if one of its child processes should

crash, that all other child processes are also stopped in a controlled way and then restarted. This ensures that the entire database maintains a consistent state in case of failure. The child processes consist of the following:

lock	A Finite-State-Machine that tracks all reading and writing processes to the database. Writes must wait for all reads to finish. Any reads that come after a write must wait for the write to finish. Currently, the entire database is locked by a writing process. This will eventually be changed to document or node-range granularity.
nodes	A file containing every XML node in the database as a 13-byte wide portion of the file in document order. The structure of this file is described in Section 3.1.
names	A simple key/value store for a 19-bit integer to a tuple containing local-name and prefix.
namespaces	A simple key/value store for a 10-bit integer to xs:anyURI.
namespace nodes	Handles the position and scope of namespace nodes in XML documents in the database. Namespace nodes are not kept in the node table.
texts	Handles a "string table" (explained in depth in Section 3.2) for all text node values in the database.
attributes	Identical to the texts process but handles attribute values instead of text nodes.
resources	A file holding all resources inserted into the database. A resource can be any unparsed text or binary content.
JSON	Holds pre-parsed JSON documents in a bit-packed binary format.
paths	A table containing each document or resource name in the database, its type, and a pointer to its location in its respective process and file. The path table is explained in greater detail in Section 3.3. The document types in the database are: 1. XML - pre-parsed XML documents 2. JSON - bit-width binary streams that when materialized make up valid JSON 3. Resource 4. Link - Linked files that are outside the database. These can be files too large to reasonably include in the resource table or files that do not exist at the time of input but will exist when first queried.

5. Item - Any XDM value (including functions). All values are serialized to a binary format and put in the Resource Table.

All the processes have access to one or more files on the filesystem to save their data and state and ensure persistency in case of failure. Processes that are expected to have a high level of write actions also have a "log file" to relieve pressure on the main data file and avoid possible file corruption. This file is used to collect updates to the main file. After a predefined number of additions to this file a process is triggered to commit the changes to the main file. Should a process crash before the log file is committed to the main file it will be committed when the process starts again. This form of buffering file changes is similar to the Journaling File System [5] used in many Linux filesystems.

3.1. Node Table

The node table contains information about every XML node in the database. Single nodes are stored as fixed-width, 13-byte binary values. These values are all stored in document order. This format allows for efficient linear scanning of all nodes for certain values and allows for fast rebuilding of contiguous nodes and node hierarchies. A mapping of all node values is shown in Table 1. The width of each value is given in bits in the column header. Each node kind has one 32-bit value that is ignored and not included in the binary value. Values of N/A mean that the value is set to all 0 value bits. This layout leans partially on the node table layout used within BaseX. [6]

Table 1. Node bit-patterns

Kind (3)	Text (32)	Offset (32)	Size (32)	Name (19)	Namespace (10)	NS Node (1)	Attributes (7)
Document	String ID of document-uri	[null]	Count of child nodes	N/A	N/A	N/A	N/A
Element	[null]	Offset to parent node	Count of attributes and child nodes	Node name ID	Namespace ID	Flag to indicate a new in-scope namespace	Count of Attributes

Kind (3)	Text (32)	Offset (32)	Size (32)	Name (19)	Name-space (10)	NS Node (1)	Attrib-utes (7)
Text	String ID of string value	Offset to parent node	[null]	N/A	N/A	N/A	N/A
Attrib-ute	String ID of attrib-ute value	N/A	[null]	Node name ID	Name-space ID	N/A	Offset to parent node
Com-ment	String ID of string value	Offset to parent node	[null]	N/A	N/A	N/A	N/A
Process-ing-instructi on	String ID of string value	Offset to parent node	[null]	Node name ID	N/A	N/A	N/A

3.2. String Table

Currently, the string table is the most complex portion of the database. It consists of five files; three data files, and two journal files. The string table is a Hash Table key/value store and strings are only stored once in the database regardless of the count of occurrences in any documents. The string ID values are limited to a 32-bit integer value. The first 28-bits of the identifier is the hash value of the string itself, the last 4 bits are a pointer to the string in an overflow array with a maximum of 16 values. Should more than 16 string values share the same hash value, an error occurs.

The decision to use a hash value as part of the key was made to allow for fast lookups of values and the fact that many databases may share the same string values. All databases will at least have the same first 28 bits in the ID for the same value. This also can also help queries that access more than one database to eliminate certain databases from selection based on a hash value not existing in the database at all. This causes the string table to act as a kind of Bloom filter [3], eliminating unnecessary disk access and lookups.

Using a fixed size key and a maximum overflow count limits the total possible capacity of the string table. To test the total capacity of a string table and effectiveness of the hashing algorithm, a simple test was run. An attempt was made to insert every possible four-byte value into a string table as if it were a normal string. This was run until the first hash collision occurred that caused a hash value to overflow the 16 possible positions. Once this error occurred, the total number of strings inserted was returned as was the total count of values per posi-

tion in the overflow array. The total count of strings at the time of error was 655,871,412. The counts per position shows how many lookups must be made in each hash bucket position array to find a certain value. As can been seen in Figure 2, the large majority of values could be found in the first four positions of the overflow array.

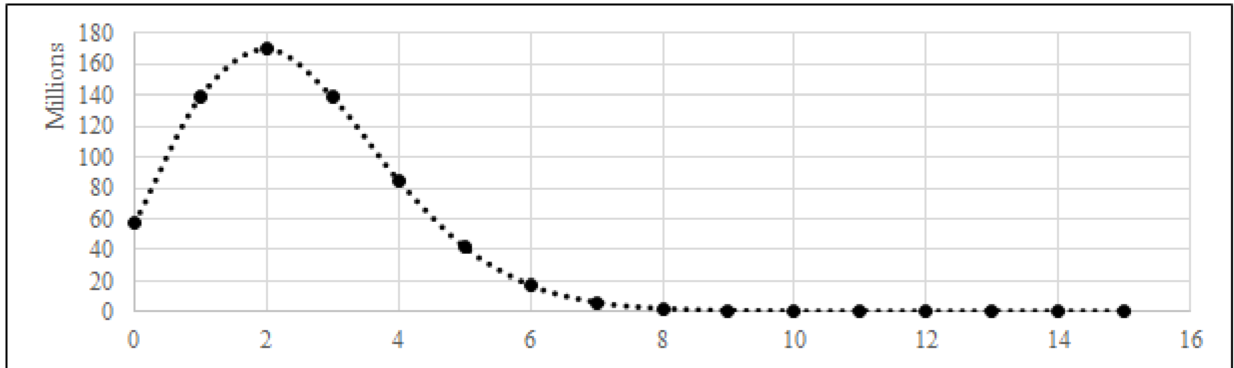


Figure 2. Unique Strings per Overflow Count at Full Capacity

To avoid overly large datafiles and wasted space when a string table is sparsely populated, the addressing of the 28-bit hash is split into three levels. The levels and sizes can be seen in Figure 3. Each level contains a pointer to the next levels position in the "index" file or in the case of the last level, a pointer into a "leaf" file that makes up the 16 overflow positions for each hash value. Only values in use at a certain level are created and written to disk. The 16 possible positions in the overflow array are split into four small blocks of pointers that are also only created by the first usage. Should a string be 11 bytes or less in length it is stored in the position block directly. Should the string be longer than 11 bytes, it is written to an append-only file called a "heap" and the position and length of the string stored.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Header Pointer												Level 1								Level 2								Position			

Figure 3. Bit Breakdown of 32-bit String ID

3.3. Path Table and Transactions

The path table is perhaps the most important part of a database as it not only provides a mapping between an available document's name and its current location on disk, but also acts as a transaction manager. This portion of the database acts as a window to all currently "committed" documents in the database. A document is only truly available to queries if it has a current record in the path table. Also, if a document should be deleted, it is only truly gone if the path table has no reference to its position on file. Changes to the path table are always the last thing

to happen in any updating query and are only visible after the database lock has been released. This form of transaction management, though somewhat crude, is effective in ensuring consistency among all parallel queries to any given database even in the case of failure.

It is worth noting that all database processes other than the path table are append-only. That is, no data is ever deleted from disk and space is not reclaimed or reused. Documents are simply dereferenced by the path table and therefore no part of them can be returned to a query.

4. Future Work

Though data can be input into, updated in and retrieved from xqerl_db, it is far from complete. The next planned additions are to implement indexing, improve XPath rewriting rules, and to allow for external backends to be queried directly from xqerl as if they were collections or databases internal to the system.

Currently, no indexes exist for data that is input into xqerl_db. The major difficulty with including indexes is the overall lack of existing open-source, persistent, sorted data stores for Erlang. There are many in-memory implementations, but only few save to disk. The few that do use persistency are either quite out of date or do not easily allow for variable-length indexed values as is needed with string values. A likely option going forward is to make a new implementation of an AA-Tree [4] that can be saved to file. This algorithm is already in use in the standard Erlang/OTP release, but only an in-memory implementation.

Changing the way that XPath expressions are handled statically and dynamically will also dramatically improve performance. In its current state, xqerl expects an in-memory version of all nodes and handles path expressions recursively as function calls on the node tree. This forces calls to the `fn:collection` and `fn:doc` functions to hold entire documents in memory. This will be changed in two ways. Firstly, in the static phase of query compilation, all XPath statements will be simplified as much as possible. Reverse axes that can be changed to a forward axis will be changed, and redundant path expressions removed, in a way as described by Olteanu et al. [2] This could eliminate the need for keeping parent and sibling nodes in memory or being built at all. Secondly, the simplified path expressions from the static phase will be sent to the individual databases that they query at runtime using a simple syntax. The database will then be responsible for finding the best query plan to execute based on the cost of execution for that expression. This will allow different databases to execute different, and possibly, best plan at the given moment instead of the query plan being decided at compile time with no knowledge of the queried data. How the cost should be calculated for any given statement is still unknown. It is also unknown how complex joins and complex predicate statements will be handled when they cross database boundaries.

Though xqerl_db will remain the default storage option, it is planned to add other storage and query options. An optional, mnesia-based backend will become part of the xqerl project and will be able to leverage the transactional, and distributed features of mnesia (the distributed database that ships with Erlang/OTP). This will be able to either be disk-based or entirely in-memory as well as optionally replicated across multiple nodes.

Other backends could also be added later to either act as the entire data source for a given node, or as a single database. These external databases could be anything from a relational database storing XML or JSON to a NoSQL database. All marshalling and unmarshalling of data will happen in the backend itself and it will be responsible for translating path expressions to the appropriate query language for the data source. This use of external data sources could vastly increase the flexibility of xqerl and the types of data it uses.

5. Conclusion

This paper has described the overall architecture of the current state of xqerl_db and some of its internal working as well as the next features to be implemented. Though far from complete, with the features mentioned above, xqerl will become an extremely flexible and dynamic XQuery processor.

Bibliography

[1] Nicola, M. John, J. *XML Parsing:* A Threat to Database Performance, CIKM 2003

[2] Olteanu, D. Meuss, H. Furche, T. Bry, F. *XPath: Looking Forward*, Institute for Computer Science and Center for Information and Language Processing University of Munich, Germany, 2001

[3] Bloom, B. H. *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, Volume 13, Issue 7 , July 1970, 422-426

[4] Andersson, A. *Balanced Search Trees Made Simple*, `http://user.it.uu.se/~arnea/ps/simp.pdf`, Retrieved 29 Jan 2019

[5] Jones, M. T. *Anatomy of Linux journaling file systems* `https://www.ibm.com/developerworks/linux/library/l-journaling-filesystems/index.html`, Retrieved 24 Jan 2019

[6] *Node Storage - BaseX Documentation*, `http://docs.basex.org/wiki/Node_Storage`, Retrieved 27 Jan. 2019

An XSLT compiler written in XSLT: can it perform?

Michael Kay
Saxonica
<mike@saxonica.com>

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

Abstract

This paper discusses the implementation of an XSLT 3.0 compiler written in XSLT 3.0. XSLT is a language designed for transforming XML trees, and since the input and output of the compiler are both XML trees, compilation can be seen as a special case of the class of problems for which XSLT was designed. Nevertheless, the peculiar challenges of multi-phase compilation in a declarative language create performance challenges, and much of the paper is concerned with a discussion of how the performance requirements were met.

1. Introduction

Over the past 18 months we have been working on a new compiler for XSLT, written in XSLT itself: see [1], [2]. At the time of writing, this is nearing functional completeness: it can handle over 95% of the applicable test cases in the W3C XSLT suite. In this paper we'll give a brief outline of the structure of this compiler (we'll call it XX), comparing and constrasting with the established Saxon compiler written in Java (which we will call XJ). And before we do that, we'll give a reminder of the motivation for writing it, from which we can derive some success criteria to decide whether it is fit for release.

Having got close to functional completeness, we now need to assess the compiler's performance, and the main part of this paper will be concerned with the process of getting the compiler to a point where the performance requirements are satisfied.

Because the compiler is, at one level, simply a fairly advanced XSLT 3.0 stylesheet, we hope that the methodology we describe for studying and improving its performance will be relevant to anyone else who has the task of creating performant XSLT 3.0 stylesheets.

2. Motivation

When XSLT 1.0 first emerged in 1999, at least a dozen implementations appeared within a year or two, many of them of excellent quality. Each typically targeted one particular platform: Java, Windows, Python, C, browsers, or whatever. Whatever your choice of platform, there was an XSLT 1.0 processor available (although on the browsers in particular, it took a few years before this goal was achieved).

For a variety of reasons, the W3C's goal of following up XSLT 1.0 with a quick 1.1 upgrade didn't happen, and it was over seven years before XSLT 2.0 came along, followed by a ten year wait for XSLT 3.0. By this time there was a sizeable XSLT user community, but very few of the original XSLT 1.0 vendors had an appetite for the development work needed to implement 2.0 or 3.0. By this stage the number of companies still developing XSLT technology was down to three: Altova and Saxonica, who both had commercial products that brought in enough revenue to fund further development, and a startup, Exselt, which had aspirations to do the same.

This pattern is not at all unusual for successful programming languages. If you look at any successful programming language, the number of implementations has declined over time as a few "winners" have emerged. But the effect of this is that the implementations that remain after the market consolidates come under pressure to cover a broader range of platforms, and that is what is happening with XSLT.

The bottom line is: there is a demand and an opportunity to deliver an XSLT processor that runs on a broader range of platforms. Over the past few years Saxon has slowly (and by a variety of bridge technologies) migrated from its original Java base to cover .NET, C, and Javascript. Currently we see demand from Node.js users. We're also having to think about how to move forward on .NET, because the bridge technology we use there (IKVM) is no longer being actively developed or maintained.

The traditional way to make a programming language portable is to write the compiler in its own language. This was pioneered by Martin Richards with BCPL in the late 1960s, and it has been the norm ever since.

Many people react with a slight horror to the idea of writing an XSLT compiler in XSLT. Surely a language that is mainly used for simple XML-to-HTML conversion isn't up to that job? Well, the language has come on a long way since version 1.0. Today it is a full functional programming language, with higher order functions and a rich set of data types. Moreover, XSLT is designed for performing transformations on trees, and transforming trees is exactly what a compiler does. So the language ought to be up to the job, and if it isn't then we would like to know why.

As we submit this paper, we have produced an almost-complete working XSLT compiler in XSLT 3.0, without encountering any serious obstacles in the lan-

guage that made the task insuperable. We'll give an outline description of how it works in the next section. But the challenging question when we started was always going to be: will it perform? Answering that question is the main purpose of this paper.

Back in 2007, Michael Kay gave a paper on writing an XSLT optimizer in XSLT: see [3]. At that time, one conclusion was that tree copying needed to be much more efficient; the paper gave an example of how a particular optimization rewrite could only be achieved by an expensive copying operation applied to a complete tree. Many optimizations are likely to involve recursive tree rewrites which perform copying of the tree; there is a serious need to optimize this design pattern.

At XML Prague 2018 (see [4]) the same author returned to this question of efficient copying of subtrees, with a proposal for new mechanisms that would allow subtrees to be virtually copied from one tree to another. One of the things examined in this paper is how much of a contribution this makes to the performance of the XSLT compiler (spoiler: the results are disappointing).

3. The Compilers

In this section we will give an outline description of two XSLT compilers: the traditional Saxon compiler, written in Java, which for the purposes of this paper we will call XJ (for "XSLT compiler written in Java"), and the new compiler, which we will call XX (for "XSLT compiler written in XSLT").

Both compilers take as input a source XSLT stylesheet (or more specifically in XSLT 3.0 a source XSLT package, because XSLT 3.0 allows packages to be compiled independently and then subsequently linked to form an executable stylesheet), and both are capable of producing as output an SEF file, which is essentially the compiled and optimized expression tree, serialized in either XML or JSON. The expression tree can then form the input to further operations: it can be directly interpreted, or executable code can be generated in a chosen intermediate or machine language. But we're not concerned in this paper with how it is used, only with how it is generated. The SEF file is designed to be portable. (We have made a few concessions to optimize for a particular target platform, but that should really be done as a post-processing phase.)

3.1. The XJ Compiler

In this section we will give an outline description of how the traditional XSLT compiler in Saxon (written in Java) operates. This compiler has been incrementally developed over a period of 20 years since Saxon was first released, and this description is necessarily an abstraction of the actual code.

It's conventional to describe a compiler as operating in a sequence of phases, even if the phases aren't strictly sequential, and I shall follow this convention. The main phases of the XJ compiler are as follows:

- The XSLT source code is processed using a standard SAX parser to produce a sequence of events representing elements and attributes.
- The content handler that receives this stream of events performs a number of operations on the events before constructing a tree representation of the code in memory. This can be regarded as a pre-processing phase. The main operations during this phase (which operates in streaming mode) are:
  - Static variables and parameters are evaluated
  - Shadow attributes are expanded into regular attributes
  - `use-when` expressions are evaluated and applied
  - `xsl:include` and `xsl:import` declarations are processed.
  - Whitespace text nodes, comments, and processing instructions are stripped.

  The result of this phase is a set of in-memory trees, one for each module in the stylesheet package being compiled. These trees use the standard Saxon "linked tree" data structure, a DOM-like structure where element nodes are represented by custom objects (subclassing the standard `Element` class) to hold properties and methods specific to individual XSLT elements such as `xsl:variable` and `xsl:call-template`.
- Indexing: the top-level components in the stylesheet (such as global variables, named templates, functions, and attribute sets) are indexed by name.
- Attribute processing: for each element in the stylesheet, the attributes are validated and processed as appropriate. This is restricted to processing that can be carried out locally. Attributes containing XPath expressions and XSLT patterns, and other constructs such as type declarations, are parsed at this stage; the result of parsing is an in-memory expression tree.
- Contextual validation: elements are validated "in context" to ensure that they appear in the proper place with the proper content model, and that consistency rules are satisfied. Also during this phase, the first type-checking analysis is carried out, confined to one XPath expression at a time. Type checking infers a static type for each expression and checks this against the required type. If the inferred type and the required type are disjoint, a static error is reported. If the required type subsumes the inferred type, all is well and no further action is needed. If the inferred type overlaps the required type, runtime type checking code is inserted into the expression tree.
- Expression tree generation (referred to, rather unhelpfully, as "compiling"). This phase changes the data representation from the decorated XDM tree used

so far to a pure tree of Java objects representing instructions and expressions to be evaluated. At this stage the boundaries between XSLT and XPath constructs disappear into a single homogenous tree; it becomes impossible to tell, for example, whether a conditional expression originated as an XPath `if-then-else` expression or as an XSLT `xsl:if` instruction.

- A second type-checking phase follows. This uses the same logic as the previous type-checking, but more type information is now available, so the job can be done more thoroughly.
- Optimization: this optional phase walks the expression tree looking for rewrite opportunities. For example, constant expressions can be evaluated eagerly; expressions can be lifted out of loops; unnecessary sort operations (of nodes into document order) can be eliminated; nested-loop joins can be replaced with indexed joins.
- When XSLT 3.0 streaming is in use, the stylesheet tree is checked for conformance to the streamability rules, and prepared for streamed execution. There is also an option to perform the streamability analysis prior to optimization, to ensure strict conformance with the streaming rules in the language specification (optimization will sometimes rewrite a non-streamable expression into a streamable form, which the language specification does not allow).
- Finally, a stylesheet export file (SEF file) may be generated, or Java bytecode may be written for parts of the stylesheet.

Some of these steps by default are deferred until execution time. When a large stylesheet such as the DocBook or DITA stylesheets is used to process a small source document, many of the template rules in the stylesheet will never fire. Saxon therefore avoids doing the detailed compilation and optimization work on these template rules until it is known that they are needed. Bytecode generation is deferred even longer, so it can focus on the hot-spot code that is executed most frequently.

The unit of compilation is an XSLT package, so there is a process of linking together the compiled forms of multiple packages. Currently a SEF file contains a package together with all the packages it uses, expanded recursively. The SEF file is a direct serialization of the expression tree in XML or JSON syntax. It is typically several times larger than the original XSLT source code. [1]

[1]SEF files generated by the XX compiler are currently rather larger than those generated by XJ. This is partly because XJ has a more aggressive optimizer, which tends to eliminate unnecessary constructs (such as run-time type checks) from the expression tree; and partly because XX leaves annotations on the SEF tree that might be needed in a subsequent optimization phase, but which are not used at run-time. The SEF representation of the XX compiler as produced by XJ is around 2Mb in expanded human-readable XML form; the corresponding version produced by XX is around 6Mb.

3.2. The XX Compiler

The XSLT compiler written in XSLT was developed as a continuation of work on adding dynamic XPath functionality to Saxon-JS ([1])). That project had constructed a robust XPath expression compiler, supporting most of the XPath 3.1 functionality, with the major exception of higher-order functions. Written in JavaScript, it generated an SEF tree for subsequent evaluation within a Saxon-JS context, and in addition determined the static type of the results of this expression.

Given the robustness of this compiler, we set about seeing if an XSLT compiler could be written, using XSLT as the implementation language and employing this XPath compiler, to support some degree of XSLT compilation support within a browser-client. Initial progress on simpler stylesheets was promising, and it was possible to run (and pass!) many of the tests from the XSLT3 test suites. We could even demonstrate a simple interactive XSLT editor/compiler/executor running in a browser. Details of this early progress and the main aspects of the design can be found in [2])

Progress was promising, but it needed a lot of detailed work to expand the functionality to support large areas of the XSLT specification correctly. For example issues such as tracking `xpath-default-namespaces`, namespace-prefix mappings and correctly determining import precedence have many corner cases that, whilst possibly very very rare in use, are actually required for conformance to the XSLT3.0 specification.

At the same time, the possibility of using the compiler within different platform environments, most noteably Node.js, increased the need to build to a very high degree of conformance to specification, while also placing demands on usability (in the form of error messages: the error messages output by a compiler are as important as the executable code), and tolerable levels of both compiling and execution performance. Performance is of course the major topic of this paper, but the work necessary to gain levels of conformance took a lot longer than might originally have been supposed, and work on usability of diagnostics has really only just started. The methodology used had two main parts:

- Checking the compiler against test-cases from the XSLT-3.0 test suite. This was mainly carried out within an interactive web page (running under Saxon-JS) that permitted tests to be selected, run, results checked against test assertions and intermediate compilation stages examined. For example the earliest work looked at compiling stylesheets that used the `xsl:choose` instruction and iteratively coding until all the thirty-odd tests were passing.
- At a later stage, the compiler had advanced sufficiently that it became possible to consider it compiling its own source, which whilst not a sufficient condition is certainly a necessary one. The test would be that after some 3-4 stages of self-compilation, the compiled-compiler 'export' tree would be constant. This

was found to be very useful indeed — for example it uncovered an issue where template rules weren't being rank-ordered correctly, only at the third round of self-compilation.

In this section we'll start by briefly discussing the (top-level) design of the compiler, but will concentrate more on considering the compiler as a program written in XSLT, before it was 'performance optimised'.

In drawing up the original design, a primary requirement was to ease the inevitable and lengthy debugging process. Consequently the design emphasised visibility of internal structures and in several parts used a multiplicity of result trees where essential processing could perhaps have been arranged in a single pass. The top-level design has some six major sequential phases, with a complete tree generated after each stage. These were:

- The first phase, called `static`, handles inclusion/importation of all stylesheet modules, together with XSLT3.0's features of static variables, conditional inclusion and shadow attributes. The result of this phase is a single XDM tree representing the merged stylesheet modules, after processing of `use-when` and shadow attributes, decorated with additional attributes to retain information that would otherwise be lost: original source code location, base URIs, namespace context, import precedence, and attributes such as `exclude-result-prefixes` inherited from the original source structure. [2]
- A normalisation phase where the primary syntax of the stylesheet/package is checked, and some normalisation of common terms (such as boolean-valued attributes 'strings', 'yes','false','0' etc), is carried out. In the absence of a full schema processor, syntax checking involves two stages: firstly a map-driven check that the XSLT element is known, has all required and no unknown attributes and has permitted child and parent elements. Secondly a series of template rules to check more detailed syntax requirements, such as `xsl:otherwise` only being the last child of `xsl:choose` and cases where either `@select` or a sequence constructor child, but not both, are permitted on an element.
- Primary compilation of the *XSLT* declarations and instructions. This phase converts the tree from the source XSLT vocabulary to the SEF vocabulary. This involves collecting a simple static context of declaration signatures (user functions, named templates) and known resources (keys, accumulators, attribute sets, decimal formats) and then processing each top level declaration to produce the necessary SEF instruction trees by recursive push processing, using the static context to check for XSLT-referred resource existence. Note that dur-

[2]We are still debating whether there would be benefits in splitting up this monolithic tree into a "forest" of smaller trees, one for each stylesheet component.

ing this phase XPath expressions and patterns are left as specific single pseudo-instructions for processing during the next phase[3].

- Compilation of the *XPath* and *pattern* expressions, and type-checking of the consequent bindings to variable and parameter values. In this phase the pseudo-instructions are compiled using a `saxon:compile-XPath` extension function, passing both the expression and complete static context (global function signatures, global and local variables with statically determined types, in-scope namespaces, context item type etc.), returning a compiled expression tree and inferred static type. These are then interpolated into the compilation tree recursively, type-checking bindings from the the XPath space to the XSLT space, i.e. typed XSLT variables and functions.

  For pragmatic reasons, the XPath parsing is done in Java or Javascript, not in XSLT. Writing an XPath parser in XSLT is of course possible, but we already had parsers in Java and Javascript, so it was easier to continue using them.

- Link-editing the cross-component references in a *component-binding* phase. References to user functions, named templates, attribute sets and accumulators needed to be resolved to the appropriate component ID and indirected via a binding vector attached to each component[4]

  . After this point the SEF tree is complete and only needs the addition of a checksum and serialization into the final desired SEF file.

Each of these phases involves a set of XSLT template rules organized into one major mode (with a default behaviour of `shallow-copy`), constructing a new result tree, but often there are subsidiary modes used to process special cases. For example, a compiled XPath expression that refers to the (function) `current()` is converted to a `let` expression that records the context item, with any reference to `current()` in the expression tree replaced with a reference to the `let` variable.

The code makes extensive use of tunnel parameters, and very little use of global variables. Indexes (for example, indexes of named templates, functions, and global variables in the stylesheet being compiled) are generally represented using XSLT 3.0 maps held in tunnel parameters.

It's worth stating at this point that the compiler currently does not use a number of XSLT3.0 features at all, for example attribute sets, keys,accumulators, `xsl:import`, schema-awareness, streaming, and higher-order functions. One reason for this was to make it easier to bootstrap the compiler; if it only uses a subset of the language, then it only needs to be able to compile that subset in order to compile itself. Late addition of support for higher-order functions in the XPath compiler makes the latter a distinct possibility, though in early debugging they

[3]In theory XPath compilation could occur during this phase, but the complexity of debugging ruled this out until a very late stage of optimisation.

[4]This derives from combination of separately-compiled packages, where component internals need not be disturbed.

may have been counter-productive. It should also be noted that separate package compilation is not yet supported, so `xsl:stylesheet`, `xsl:transform` and `xsl:package` are treated synonymously.

A run of the compiler can be configured to stop after any particular stage of this process, enabling the tree to be examined in detail.

We'll now discuss this program not as an XSLT compiler, but as an example of a large XSLT transformation, often using its self-compilation as a sample stress-testing workload.

The XX compiler is defined in some 33 modules, many corresponding to the relevant section of the XSLT specification. Internally there is much use of static-controlled inclusion (`@use-when`) to accommodate different debugging, operational and optimisation configurations, but when this phase has been completed, the program (source) tree has some 536 declarations, covering 4,200 elements and some 7,200 attributes, plus another 13,500 attributes added during inclusion to track original source properties, referred to above. The largest declaration (the template that 'XSLT-compiles' the main stylesheet) has 275 elements, the deepest declaration (the primary static processing template) is a tree up to 12 elements deep.

Reporting of syntax errors in the user stylesheet being compiled is currently directed to the `xsl:message` output stream. Compilation continues after an error, at least until the end of the current processing phase. The relevant error-handling code can be overridden (in the usual XSLT manner) in a customization layer to adapt to the needs of different platforms and processing environments.

Top level processing is a chain of five XSLT variables bound to the push ('apply-templates') processing of the previous (tree) result of the chain. We'll examine each of these in turn:

3.2.1. Static inclusion

The XSLT architecture for collecting all the relevant sections of the package source is complicated mainly by two features: firstly the use of static global variables as a method of *meta-programming*, controlling conditional source inclusion, either through `@use-when` decorations or even through *shadow attributes* on inclusion/importation references. Secondly it is critical to compute the import precedence of components, which requires tracking importation depth of the original source. Other minor inconveniences include the possibility of the XSLT version property changing between source components and the need to keep track of original source locations (module names and line numbers).

As static variables can only be global (and hence direct children of a stylesheet) and their scope is (almost) `following-sibling::*/ descendant-or-self::*`, the logic for this phase needs to traverse the top-level sibling declarations maintaining state as it goes (to hold information about the static vari-

ables encountered. The XSLT 3.0 `xsl:iterate` instruction is ideally suited to this task. The body of the `xsl:iterate` instruction collects definitions of static variables in the form of a map. Each component is then processed by template application in mode `static`, collecting the sequence of processed components as a parameter of the iteration. Static expressions may be encountered as the values of static variables, in `[xsl:]use-when` attributes, and between curly braces in shadow attributes; in all cases they are evaluated using the XSLT 3.0 `xsl:evaluate` instruction, with in-scope static variables supplied as the `@with-params` property.[5]The result of the evaluation affects subsequent processing:

- For `[xsl:]use-when`, the result determines whether the relevant subtree is processed using recursive `xsl:apply-templates`, or discarded
- For static variables and parameters, the result is added to a map binding the names of variables to their values, which is made available to following sibling elements as a parameter of the controlling `xsl:iterate`, and to their descendant instructions via tunnel parameters.
- For shadow attributes, the result is injected into the tree as a normal (non-shadow) attribute. For example the shadow attribute `_streamable="{$STREAMING}"` might be rewritten as `streamable="true"`.

Errors found during the evaluation of static XPath expressions will result in exceptions during `xsl:evaluate` evaluation - these are caught and reported.

After each component has been processed through the `static` phase, it is typically added to the `$parts` parameter of the current iteration. In cases where the component was the declaration of a static variable or parameter, the `@select` expression is evaluated (with `xsl:evaluate` and the current bindings of static variables) and its binding added to the set of active static variables.

Processed components which are `xsl:include|xsl:import` declarations are handled within the same iteration. After processing the `@href` property is resolved to recover the target stylesheet[6]. The stylesheet is then read and processed in the `static` mode. The result of this a map with two members — the processed components and the number of prior imports. The processed components are then allocated an importation precedence (recorded as an attribute) dependent upon importation depth/position and any previous precedence and added to the set of components of the including stylesheet[7]. Finally the complete

[5]There is a minor problem here, in that `use-when` expressions are allowed access to some functions, such as `fn:system-property()`, which are not available within `xsl:evaluate`. In a few cases like this we have been obliged to implement language extensions.

[6]A stack of import/included stylesheets is a parameter of the main stylesheet template, the check against self or mutual recursive inclusion.

[7]This complexity is due to the possibility of an importation, referenced via an inclusion, preceding a high-level importation - something permitted in XSLT3.0. Note that the current XX compiler does not itself use `xsl:import` - linkage is entirely through `xsl:include`.

sequence of self and included components are returned as a map with the 'local' importation information. At the very top level the stylesheet is formed by copying all active components into the result tree.

In more general XSLT terms, the processing involves recursive template application for the entire (extended) source tree, with stateful iteration of the body of stylesheets, evaluation and interpolation of static variables with that iteration and a complex multiple-copy mechanism for recording/adjusting importation precedence.

3.2.2. Normalisation

The normalisation phase makes intensive use of XSLT template rules. Generally, each constraint that the stylesheet needs to satisfy (for example, that the `type` and `validation` attributes are mutually exclusive) is expressed as a template rule. Each element in the use stylesheet is processed by multiple rules, achieved by use of the `xsl:next-match` instruction.

The normalisation phase has two main mechanisms. The first involves checking any `xsl:*` element for primary syntax correctness — is the element name known, does it have all required attributes or any un-permitted attributes, do any 'typed' attributes (e.g. boolean) have permitted values and are parent/child elements correct? A simple schema-like data structure[8] was built from which a map *element-name => {permitted attributes, required attributes, permitted parents, permitted children...}* was computed, and this is used during the first stage of syntax checking through a high-priority template. The second mechanism is more ad-hoc, and comprises a large set of templates matching either error conditions such as:

```
<xsl:template match="xsl:choose[empty(xsl:when)]" mode="normalize">
    <xsl:sequence select="f:missingChild(., 'xsl:when')"/>
</xsl:template>
```

which checks that a 'choose' must have a when 'clause', or normalising a value, such as:

```
<xsl:template match="xsl:*/@use-attribute-sets" mode="normalize">
   <xsl:attribute name="use-attribute-sets"
                  select="tokenize(.) ! f:EQName(., current()/..)"/>
</xsl:template>
```

which normalises attribute set names to EQNames.

As far as XSLT processing is concerned, this phases builds one tree in a single pass over the source tree.

[8]Derived from the syntax definitions published with the XSLT3.0 specification

3.2.3. XSLT compilation

The main compilation of the XSLT package involves three main processes — collecting (properties of) all the global resources of the package, such as named templates, user-defined functions, and decimal formats; collecting all template rules into same-mode groups; and a recursive descent compilation of XSLT instructions of each component.

The process for the first is to define a set of some dozen variables, which are then passed as tunnel parameters in subsequent processing, such as:

```
<xsl:variable name="named-template-signatures" as="map(*)">
  <xsl:map>
    <xsl:for-each-group select="f:precedence-sort(xsl:template)"
                        group-by="@name">
      <xsl:variable name="highest" select="
              let $highest-precedence :=
                    max(current-group()/@ex:precedence)
              return
                current-group()[@ex:precedence = $highest-precedence]"/>
      <xsl:if test="count($highest) gt 1">
        <xsl:sequence select="f:syntax-error('XTSE0660',
            'Multiple declarations of ' || name() || ' name=' || @name ||
            ' at highest import precedence')"/>
      </xsl:if>
      <xsl:variable name="params"
                  select="$highest/xsl:param[not(@tunnel eq 'true')]"/>
      <xsl:map-entry key="$highest/@name" select="map{
            'params': f:string-map($params/map:entry(@name,
                                  map{'required': @required eq 'true',
                                      'type': @as})),
            'required': $params[@required eq 'true']/@name,
            'type': ($highest/@as,'item()*')[1]
            }"/>
    </xsl:for-each-group>
  </xsl:map>
</xsl:variable>
```

which both checks for conflicting named templates, handles differing precedences and returns a map of names/signatures. This can then of course be referenced in compiling a `xsl:call-template` instruction to check both the existence of the requested template and the names/types of its parameters, as well as the implied result type.

All template rules are first expanded into 'single mode' instances by copying for each referred `@mode` token[9]

. From this all used modes can be determined and for each a mode component is constructed and populated with the relevant compiled templates. A pattern

matching template is compiled with a simple push template, that leaves the `@match` as a `pattern` pseudo-instruction, and the body as a compiled instruction tree. The design currently involves the construction of *three* trees for each template during this stage.

The bulk of the XSLT compiling is a single recursive set of templates, some of which check for error conditions[10], most of which generate an SEF instruction and recursively process attributes and children, such as:

```
<xsl:template match="xsl:if" mode="sef">
  <xsl:param name="attr" as="attribute()*" select="()"/>
  <choose>
    <xsl:sequence select="$attr"/>
    <xsl:call-template name="record-location"/>
    <xsl:apply-templates select="@test" mode="create.xpath"/>
    <xsl:call-template name="sequence-constructor"/>
    <true/>
    <empty/>
  </choose>
</xsl:template>
```

which generates a `choose` instruction for `xsl:if`, with any required attributes attached (often to identify the role of the instruction in its parent), records the source location, creates an `xpath` pseudo-instruction for the test expression, adds the sequence constructor and appends an empty 'otherwise' case.

Local `xsl:variable` and `xsl:param` instructions are replaced by `VARDEF` and `PARAMDEF` elements for processing during XPath compiling.

The final result of this phase is a `package` with a series of component children corresponding to compiled top-level declarations and modes with their template rules.

3.2.4. XPath compiling and type checking

In this phase the package is processed to compile the `xpath` and `pattern` pseudo-instructions, determine types of variables, parameters, templates and functions and propogate and type-check cross-references. As such the key action is an iteration through the children of any element that contains `VARDEF` or `PARAMDEF` children, accumulating variable/type bindings that are passed to the XPath compiler. Unlike the similar process during the static phase, in this case the architecture is to use a recursive named template, to build a nested tree of `let` bindings, propogating variable type bindings downwards and sequence constructor result types back upwards. In this case the result type is held as an `@sType` attribute value. The

[9]This has the unfortunate effect of duplicating bodies (and compilation effort thereof) for multi-mode templates — an indexed design might be a possibility, but may require SEF additions

[10]And perhaps should exist in in the normalisation phase

top of this process determines the type of a component's result, which can be checked against any declaration (I.e.`@as`)

This phase requires a static type system and checker which generates a small map structure (*baseType, cardinality.....*) from the XSchema string representation and uses this to compare supplied and required types, determining whether there is match, total conflict or a need for runtime checking. Written in XSLT, one drawback is that the type of result trees is returned as a string on an attribute, requiring reparsing[11].

Some instructions require special processing during this phase. Some, e.g. `forEach`, alter the type of the context item for evaluation of their sequence constructor body. Others, such as `choose`, return a result whose type is the union of those of their 'action' child instructions. These are handled by separate templates for each case.

Finally the `pattern` instructions are compiled. For accumulators and keys their result trees are left on their parent declaration. For template rules, in addition, the default priority of the compiled pattern is calculated if required and with a priority and import precedence available for every template rule in a mode, they can be rank ordered.

3.2.5. Component binding

At this point all the compiling is complete, but all the cross-component references must be linked. This is via a two stage process: firstly building a component 'name' to component number ('id') map. Then each component is processed in turn, collecting all descendant references (`call-template`, user-function calls, key and accumulator references etc.) and building an indirect index on the component head, whose entries are then interpolated into the internal references during a recursive copy.[12]

3.2.6. Reflections on the design

We must emphasise that this architecture was designed for ease of the (complex) debugging anticipated, valuing visibility over performance. Several of the phases could be coalesced, reducing the need for multiple copying of large trees. For example the normalisation and the compiling phases could be combined into a single set of templates for each XSLT element, the body of which both checked

[11]Changing the canonical return to a (map) tuple of *(tree,type)* could be attempted but it would make the use of a large set of element-matching templates completely infeasible.

[12]In XSLT 2.0, all references to components such as variables, named templates, and functions could be statically resolved. This is no longer the case in XSLT 3.0, where components (if not declared private or final) can be overridden in another stylesheet package, necessitating a deferred binding process which in Saxon is carried out dynamically at execution time. The compiler generates a set of binding vectors designed to make the final run-time binding operation highly efficient.

syntax and compiled the result[13]. Similarly the XSLT and XPath compilation phases could be combined, incorporating static type checking in the same operation. And some ot the operations, especially in type representation, may be susceptible to redesign. Some of these will be discussed in the following sections

3.3. Comparing the Two Compilers

At a high level of description, the overall structure of the two compilers is not that different. Internally, the most conspicuous difference is in the internal data structures.

Both compilers work initially with the XDM tree representation of the stylesheet as a collection of XML documents, and then subsequently transform this to an internal representation better suited to operations such as type-checking.

For the XJ compiler, this internal representation is a mutable tree of Java objects (each node in the tree is an object of class `Expression`, and the references to its subexpressions are via objects of class `Operand`). The final SEF output is then a custom serialization of this expression tree. The expression tree is mutable, so there is no problem decorating it with additional properties, or with performing local rewrites that replace selected nodes with alternatives. It's worth noting, however, that the mutability of the tree has been a rich source of bugs over the years. Problems can and do arise through properties becoming stale (not being updated when they should be), through structural errors in rewrite operations (leading for example to nodes having multiple parents), or through failure to keep the structure thread-safe.

For the XX compiler, the internal representation is itself an XDM node tree, augmented with maps used primarily as indexes into the tree. This creates two main challenges. Firstly, the values of elements and attributes are essentially limited to strings; this leads to clumsy representation of node properties such as the inferred type, or the set of in-scope namespaces. As we will see, profiling showed that a lot of time was being spent translating such properties from a string representation into something more suitable for processing (and back). Secondly, the immutability of the tree leads to a lot of subtree copying. To take just one example, there is a process that allocates distinct slot numbers to all the local variable declarations in a template or function. This requires one pass over the subtree to allocate the numbers (creating a modified copy of the subtree as it goes). But worse, on completion we want to record the total number of slots allocated as an attribute on the root node of the subtree; the simplest way of achieving this is to

[13]This is something of an anathema to accepted XSLT wisdom in the general case, where a mutliplicity of pattern-matching templates is encouraged, but in this case the 'processed target', i.e. the XSLT language isn't going to change.

copy the whole subtree again. As we will see, subtree copying contributes a considerable part of the compilation cost.

4. Compiler Performance

The performance of a compiler matters for a number of reasons:

- **Usability and Developer Productivity.** Developers spend most of their time iteratively compiling, discovering their mistakes, and correcting them. Reducing the elapsed time from making a mistake to getting the error message has a critical effect on the development experience. Both the authors of this paper have been around long enough to remember when this time was measured in hours. Today, syntax-directed editors often show you your mistakes before you have finished typing. In an XML-based IDE such as oXygen, the editing framework makes repeated calls on the compiler to get diagnostics behind the scenes, and the time and resource spent doing this has a direct impact on the usability of the development tool.
- **Production efficiency.** In some environments, for example high volume transaction processing, a program is compiled once and then executed billions of times. In that situation, compile cost is amortized over many executions, so the cost of compiling hardly matters. However, there are other production environments, such as a publishing workflow, where it is common practice to compile a stylesheet each time it is used. In some cases, the cost of compiling the stylesheet can exceed the cost of executing it by a factor of 100 or more, so the entire elapsed time of the publishing pipeline is in fact dominated by the XSLT compilation cost.
- **Spin-off benefits.** For this project, we also have a third motivation: if the compiler is written in XSLT, then making the compiler faster means we have to make XSLT run faster, and if we can make XSLT run faster, then the execution time of other (sufficiently similar) stylesheets should all benefit. Note that "making XSLT run faster" here doesn't just mean raw speed: it also means the provision of instrumentation and tooling that helps developers produce good, fast code.

4.1. Methodology

Engineering for performance demands a disciplined approach.

- The first step is to set requirements, which must be objectively measurable, and must be correlated with the business requirements (that is, there must be a good answer to the question, what is the business justification for investing effort to make it run faster?)

Often the requirements will be set relative to the status quo (for example, improve the speed by a factor of 3). This then involves measurement of the status quo to establish a reliable baseline.

- Then it becomes an iterative process. Each iteration proceeds as follows:
  - Measure something, and (important but easily forgotten) keep a record of the measurements.
  - Analyze the measurements and form a theory about why the numbers are coming out the way they are.
  - Make a hypothesis about changes to the product that would cause the numbers to improve.
  - Implement the changes.
  - Repeat the measurements to see what effect the changes had.
  - Decide whether to retain or revert the changes.
  - Have the project requirements now been met? If so, stop. Otherwise, continue to the next iteration.

4.2. Targets

For this project the task we want to measure and improve is the task of compiling the XX compiler. We have chosen this task because the business objective is to improve the speed of XSLT compilation generally, and we think that compiling the XX compiler is likely to be representative of the task of compiling XSLT stylesheets in general; furthermore, because the compiler is written in XSLT, the cost of compiling is also a proxy for the cost of executing arbitrary XSLT code. Therefore, any improvements we make to the cost of compiling the compiler should benefit a wide range of other everyday tasks.

There are several ways we can compile the XX compiler (remembering that XX is just an XSLT stylesheet).

We can describe the tasks we want to measure as follows:

E0: $C_{EEJ}(XX)$ ➔ XX_0 (240ms ➔ 240ms)

Exercise E0 is to compile the stylesheet XX using the built-in XSLT compiler in Saxon-EE running on the Java platform (denoted here C_{EEJ}) to produce an output SEF file which we will call XX_0. The baseline timing for this task (the status quo cost of XSLT compilation) is 240ms; the target remains at 240ms.

"E1: $T_{EEJ}(XX, XX_0)$ ➔ XX_1 (2040ms ➔ 720ms)"

Exercise E1 is to apply the compiled stylesheet XX_0 to its own source code, using as the transformation engine Saxon-EE on the Java platform (denoted here T_{EEJ}(source, stylesheet)), to produce an output SEF file which we will call XX_1. Note that XX_0 and XX_1 should be functionally equivalent, but they are not required to be identical (the two compilers can produce different executables, so

long as the two executables do the same thing). The measured baseline cost for this transformation is 2040ms, which means that the XX compiler is 8 or 9 times slower than the existing Saxon-EE/J compiler. We would like to reduce this overhead to a factor of three, giving a target time of 720ms.

"E2: $T_{JSN}(XX, XX_0)$ ➔ XX_2 (90s ➔ 3s)"

Exercise E2 is identical, except that this time we will use as our transformation engine Saxon-JS running on Node.js. The ratio of the time for this task compared to E1 is a measure of how fast Saxon on Node.js runs relative to Saxon on Java, for one rather specialised task. In our baseline measurements, this task takes 90s – a factor of 45 slower. That's a challenge. Note that this doesn't necessarily mean that every stylesheet will be 45 times slower on Node.js than on Java. Although we've described XX as being written in XSLT, that's a simplification: the compiler delegates XPath parsing to an external module, which is written in Java or Javascript respectively. So the factor of 45 could be due to differences in the two XPath parsers. At the moment, though, we're setting requirements rather than analysing the numbers. We'll set ourselves an ambitious target of getting this task down to three seconds.

"E3: $T_{EEJ}(XX, XX_1)$ ➔ XX_3 (2450ms ➔ E1 + 25%) "

Exercise E3 is again similar to E1, in that it is compiling the XX compiler by applying a transformation, but this time the executable stylesheet used to perform the transformation is produced using the XX compiler rather than the XJ compiler. The speed of this task, relative to E1, is a measure of how good the code produced by the XX compiler is, compared with the code produced by the XJ compiler. We expected and were prepared to go with it being 25% slower, but found on measurement that we were already exceeding this goal.

There are of course other tasks we could measure; for example we could do the equivalent of E3, but using Saxon-JS rather than Saxon-EE/J. However, it's best to focus on a limited set of objectives. Repeatedly compiling the compiler using itself might be expected to converge, so that after a couple of iterations the output is the same as the input: that is, the process should be idempotent. Although technically idempotence is neither a necessary nor a sufficient condition of correctness, it is easy to assess, so as we try to improve performance, we can use idempotence as a useful check that we have not broken anything. We believe that if we can achieve these numbers, then we have an offering on Node.js that is fit for purpose; 3 seconds for a compilation of significant size will not cause excessive user frustration. Of course, this is a "first release" target and we would hope to make further improvements in subsequent releases.

4.3. Measurement Techniques

In this section we will survey the measurement techniques used in the course of the project. The phase of the project completed to date was, for the most part,

running the compiler using Saxon-EE on the Java platform, and the measurement techniques are therefore oriented to that platform.

We can distinguish two kinds of measurement: bottom-line measurement intended directly to assess whether the compiler is meeting its performance goals; and internal measurements designed to achieve a better understanding of where the costs are being incurred, with a view to making internal changes.

- The bottom-line execution figures were obtained by running the transformation from the command line (within the IntelliJ development environment, for convenience), using the `-t` and `-repeat` options.

  The `-t` option reports the time taken for a transformation, measured using Java's `System.nanoTime()` method call. Saxon breaks the time down into stylesheet compilation time, source document parsing/building time, and transformation execution time.

  The `-repeat` option allows the same transformation to be executed repeatedly, say 20 or 50 times. This delivers results that are more consistent, and more importantly it excludes the significant cost of starting up the Java Virtual Machine. (Of course, users in real life may experience the same inconsistency of results, and they may also experience the JVM start-up costs. But our main aim here is not to predict the performance users will obtain in real life, it is to assess the impact of changes we make to the system.)

  Even with these measures in place, results can vary considerably from one run to another. That's partly because we make no serious attempt to prevent other background work running on the test machine (email traffic, virus checkers, automated backups, IDE indexing), and partly because the operating system and hardware themselves adjust processor speed and process priorities in the light of factors such as the temperature of the CPU and battery charge levels. Some of the changes we have been making might only deliver a 1% improvement in execution speed, and 1% is unfortunately very hard to measure when two consecutive runs, with no changes at all to the software, might vary by 5%. Occasionally we have therefore had to "fly blind", trusting that changes to the code had a positive effect even though the confirmation only comes after making a number of other small changes whose cumulative effect starts to show in the figures.

  Generally we trust a good figure more than we trust a bad figure. There's an element of wishful thinking in this, of course; but it can be justified on the basis that random external factors such as background processes can slow a test run down, but they are very unlikely to speed it up. The best figures we got were usually when we ran a test first thing in the morning on a cold machine.

- *Profiling:* The easiest way to analyse where the costs are going for a Saxon XSLT transformation is to run with the option `-TP:profile.html`. This gener-

ates an HTML report showing the gross and net time spent in each stylesheet tempate or function, together with the number of invocations. This output is very useful to highlight hot-spots.

Like all performance data, however, it needs to be interpreted with care. For example, if a large proportion of the time is spent evaluating one particular match pattern on a template rule, this time will not show up against that template rule, but rather against all the template rules containing an `xsl:apply-templates` instruction that causes the pattern to be evaluated (successfully or otherwise). This can have the effect of spreading the costs thinly out among many other templates.

- *Subtractive measurement:* Sometimes the best way to measure how long something is taking is to see how much time you save by not doing it. For example, this technique proved the best way to determine the cost of executing each phase of the compiler, since the compiler was already structured to allow early termination at the end of any phase. It can also be used in other situations: for example, if there is a validation function testing whether variable names conform to the permitted XPath syntax, you can assess the cost of that operation by omitting the validation. (As it happens, there's a cheap optimization here: test whether names are valid at the time they are declared, and rely on the binding of references to their corresponding declarations to catch any invalid names used as variable or function references).
- A corresponding technique, which we had not encountered before this project, might be termed *additive measurement*. Sometimes you can't cut out a particular task because it is essential to the functionality; but what you can do is to run it more than once. So, for example, if you want to know how much time you are spending on discovering the base URIs of element nodes, one approach is to modify the relevant function so it does the work twice, and see how much this adds to total execution time.
- *Java-level profiling.* There's no shortage of tools that will tell you where your code is spending its time at the Java level. We use JProfiler, and also the basic `runhprof` reports that come as standard with the JDK. There are many pitfalls in interpreting the output of such tools, but they are undoubtedly useful for highlighting problem areas. Of course, the output is only meaningful if you have some knowledge of the source code you are profiling, which might not be the case for the average Saxon XSLT user. Even without this knowledge, however, one can make inspired guesses based on the names of classes and methods; if the profile shows all the time being spent in a class called `DecimalArithmetic`, you can be fairly sure that the stylesheet is doing some heavy computation using `xs:decimal` values.
- *Injected counters.* While timings are always variable from one run to another, counters can be 100% replicable. Counters can be injected into the XSLT code

by calling `xsl:message` with a particular error code, and using the Saxon extension function `saxon:message-count()` to display the count of messages by error code. Internally within Saxon itself, there is a general mechanism allowing counters to be injected: simply add a call on `Instrumentation.count("label")` at a particular point in the source code, and at the end of the run it will tell you the number of executions for each distinct label. The label does not need to be a string literal; it could, for example, be an element name, used to count visits to nodes in the source document by name. This is how we obtained the statistics (mentioned below) on the incidence of different kinds of XPath expression in the stylesheet.

  The information from counters is indirect. Making a change that reduces the value of a counter gives you a warm feeling that you have reduced costs, but it doesn't quantify the effect on the bottom line. Nevertheless, we have found that strategically injected counters can be a valuable diagnostic tool.

- *Bytecode monitoring.* Using the option -TB on the Saxon command line gives a report on which parts of the stylesheet have been compiled into Java bytecode, together with data on how often these code fragments were executed. Although it was not orginally intended for the purpose, this gives an indication of where the hotspots in the stylesheet are to be found, at a finer level of granularity than the -TP profiling output.

A general disadvantage of all these techniques is that they give you a worm's-eye view of what's going on. It can be hard to stand back from the knowledge that you're doing millions of string-to-number conversions (say), and translate this into an understanding that you need to fundamentally redesign your data structures or algorithms.

4.4. Speeding up the XX Compiler on the Java Platform

The first task we undertook (and the only one fully completed in time for publication) was to measure and improve the time taken for compiling the XX compiler, running using Saxon-EE on the Java platform. This is task E1 described above, and our target was to improve the execution time from 2040ms to 720ms.

At this stage it's probably best to forget that the program we're talking about is a compiler, or that it it is compiling itself. Think of it simply as an ordinary, somewhat complex, XML transformation. We've got a transformation defined by a stylesheet, and we're using it to transform a set of source XML documents into a result XML document, and we want to improve the transformation time. The fact that the stylesheet is actually an XSLT compiler and that the source document is the stylesheet itself is a complication we don't actually need to worry about.

We started by taking some more measurements, taking more care over the measurement conditions. We discovered that the original figure of 2040ms was obtained with bytecode generation disabled, and that switching this option on

improved the performance to 1934ms. A gain of 5% from bytecode generation for this kind of stylesheet is not at all untypical (significantly larger gains are sometimes seen with stylesheets that do a lot of arithmetic computation, for example).

Analysis of Stylesheet Execution Time

Total time: 4841.351 milliseconds

Time spent in each template, function or global variable:

The table below is ordered by the total net time spent in the template, function or global variable. Gross time means the time including called templates and functions (recursive calls only count from the original entry); net time means time excluding time spent in called templates and functions.

module	line	instruction	count	average time (gross/ms)	total time (gross/ms)	average time (net/ms)	total time (net/ms)
xpath.xsl	659	template *:xpath	2,799	0.502	1,403.761	0.438	1,227.262
component-bindings.xsl	41	template ex:co | ex:accumulator | ex:key	203	1.252	254.205	1.010	204.953
namespaces.xsl	79	function f:recover-namespaces	3,161	0.061	193.844	0.061	193.844
namespaces.xsl	56	function f:encode-namespaces	4,360	0.042	184.961	0.042	184.961
xpath.xsl	400	template process-variables	1,609	1.022	1,645.059	0.099	159.342
static.xsl	627	template staticText	4,348	0.105	456.259	0.036	158.129
type.xsl	170	function t:type	7,244	0.024	177.038	0.022	156.808
normalize.xsl	51	template *	4,341	0.101	439.124	0.035	149.904
xpath.xsl	794	template *:pattern	362	0.436	157.695	0.388	140.300

Figure 1. Example of -TP profile output

Our next step was to profile execution using the `-TP` option. Figure 1 shows part of the displayed results. The profile shows that 25% of the time is spent in a single template, the template rule with `match="*:xpath`. This is therefore a good candidate for further investigation.

4.4.1. XPath Parsing

The `match="*:xpath` template is used to process XPath expressions appearing in the stylesheet. As already mentioned, the XPath parsing is not done in XSLT code, but by a call-out to Java or Javascript (in this case, Java). So the cost of this template includes all the time spent in the Java XPath parser. However, the total time spent in this template exceeds the total cost of running the XJ compiler, which is using the same underlying XPath parser, so we know that it's not simply an inefficiency in the parser.

Closer examination showed that the bulk of the cost was actually in setting up the data structures used to represent the static context of each XPath expression. The static context includes the names and signatures of variables, functions, and types that can be referenced by the XPath expression, and it is being passed from the XSLT code to the Java code as a collection of maps. Of course, the average XPath expression (think `select="."`) doesn't use any of this information, so the whole exercise is wasted effort.

Reducing this cost used a combination of two general techniques:

- *Eager evaluation:* A great deal of the static context is the same for every XPath expression in the stylesheet: every expression has access to the same functions and global variables. We should be able to construct this data structure once, and re-use it.

- *Lazy evaluation:* Other parts of the static context (notably local variables and namespace bindings) do vary from one expression to another, and in this case the trick is to avoid putting effort into preparing the information in cases when it is not needed. One good way to do this would be through callbacks - have the XPath parser ask the caller for the information on demand (through callback functions such as a variable resolver and a namespace resolver, as used in the JAXP XPath API). However, we decided to rule out use of higher-order functions on this project, because they are not available on all Saxon versions. We found an alternative that works just as well: pass the information to the parser in whatever form it happens to be available, and have the parser do the work of digesting and indexing it only if it is needed.

These changes brought execution time down to 1280ms, a good step towards the target of 720ms.

Profiling showed that invocation of the XPath parser still accounted for a large proportion of the total cost, so we subsequently revisited it to make further improvement. One of the changes was to recognize simple path expressions like `.` and `()`. We found that of 5100 path expressions in the stylesheet, 2800 had 5 or fewer tokens; applying the same analysis to the Docbook stylesheets gave similar results. The vast majority of these fall into one of around 25 patterns where the structure of the expression can be recognised simply from the result of tokenization: if the sequence of tokens is (dollar, name) then we can simply look up a function that handles this pattern and converts it into a variable reference, bypassing the recursive-descent XPath parser entirely. Despite a good hit rate, the effect of this change on the bottom line was small (perhaps 50ms, say 4%). However, we decided to retain it as a standard mechanism in the Java XPath parser. The benefit for Java applications using XPath to navigate the DOM (where it is common practice to re-parse an XPath expression on every execution) may be rather greater.

4.4.2. Further investigations

After the initial success improving the interface to the XPath parser, the profile showed a number of things tying for second place as time-wasters: there are about 10 entries accounting for 3% of execution time each, so we decided to spread our efforts more thinly. This proved challenging because although it was easy enough to identify small changes that looked beneficial, measuring the effect was tough, because of the natural variation in bottom-line readings.

Here are some of the changes we made in this next stage of development:

- During the first ("static") phase of processing, instead of recording the full set of in-scope namespace bindings on every element, record it only if the namespace context differs from the parent element. The challenge is that there's no

easy way to ask this question in XPath; we had to introduce a Saxon extension function to get the information (`saxon:has-local-namespaces()`).

- The template rule used to strip processing instructions and comments, merge adjacent text nodes, and strip whitespace, was contributing 95ms to total execution time (say 7%). Changing it to use `xsl:iterate` instead of `xsl:for-each-group` cut this to 70ms.
- There was a very frequently executed function `t:type` used to decode type information held in string form. Our initial approach was to use a memo function to avoid repeated decoding of the same information. Eventually however, we did a more ambitious redesign of the representation of type information (see below).
- The compiler maintains a map-based data structure acting as a "schema" for XSLT to drive structural validation. This is only built once, in the form of a global variable, but when the compiler is only used for one compilation, building the structure is a measurable cost. We changed the code so that instead of building the structure programmatically, it is built by parsing a JSON file.
- We changed the code in the final phase where component bindings are fixed up to handle all the different kinds of component references (function calls, global variable references, call-template, attribute set references, etc) in a single pass, rather than one pass for each kind of component. There were small savings, but these were negated by fixing a bug in the logic that handled duplicated names incorrectly. (This theme came up repeatedly: correctness always comes before performance, which sometimes means the performance numbers get worse rather than better.)
- There's considerable use in the compiler of XSLT 3.0 maps. We realised there were two unnecessary sources of inefficiency in the map implementation. Firstly, the specification allows the keys in a map to be any atomic type (and the actual type of the key must be retained, for example whether it is an `xs:NCName` rather than a plain `xs:string`). Secondly, we're using an "immutable" or "persistent" map implementation (based on a hash trie) that's optimized to support `map:put()` and `map:remove()` calls, when in fact these hardly ever occur: most maps are never modified after initial construction. We added a new map implementation optimized for string keys and no modification, and used a combination of optimizer tweaks and configuration options to invoke it where appropriate.

Most of these changes led to rather small benefits: we were now seeing execution times of around 1120ms. It was becoming clear that something more radical would be needed to reach the 720ms goal.

At this stage it seemed prudent to gather more data, and in particular it occurred to us that we did not really have numbers showing how much time was spent in each processing phase. We tried two approaches to measuring this: one was to output timestamp information at the end of each phase, the other was "subtractive measurement" - stopping processing before each phase in turn, and looking at the effect on the bottom line. There were some interesting discrepancies in the results, but we derived the following "best estimates":

Table 1. Execution times for each phase of processing

Static processing	112ms
Normalisation	139ms
"Compilation" (generating initial SEF tree)	264ms
XPath parsing	613ms
Component binding	55ms

These figures appear to contradict what we had learnt from the `-TP` profile information. It seems that part of the discrepancy was in accounting for the cost of serializing the final result tree: serialization happens on-the-fly, so the cost appears as part of the cost of executing the final phase of the transformation, and this changes when the transformation is terminated early. It's worth noting also that when `-TP` is used, global variables are executed eagerly, so that the evaluation cost can be separated out; the tracing also suppresses some optimizations such as function inlining. Heisenberg rules: measuring things changes what you are measuring.

At this stage we decided to study how much time was being spent copying subtrees, and whether this could be reduced.

4.4.3. Subtree Copying

At XML Prague 2018, one of the authors presented a paper on mechanisms for tree copying in XSLT; in particular, looking at whether the costs of copying could be reduced by using a "tree grafting" approach, where instead of making a physical copy of a subtree, a virtual copy could be created. This allows one physical subtree to be shared between two or more logical trees; it is the responsibility of the tree navigator to know which real tree it is navigating, so that it can do the right thing when retracing its steps using the ancestor axis, or when performing other context-dependent operations such as computing the base URI or the in-scope namespaces of a shared element node.

In actual fact, two mechanisms were implemented in Saxon: one was a fast 'bulk copy" of a subtree from one TinyTree to another (exploiting the fact that both use the same internal data structure to avoid materializing the nodes in a

neutral format while copying), and the other was a virtual tree graft. The code for both was present in the initial Saxon 9.9 release, though the "bulk copy" was disabled. Both gave good performance results in synthetic benchmarks.

On examination, we found that the virtual tree grafting was not being extensively used by the XX compiler, because the preconditions were not always satisfied. We spent some time tweaking the implementation so it was used more often. After these adjustments, we found that of 93,000 deep copy operations, the grafting code was being used for over 82,000 of them.

However, it was not delivering any performance benefits. The reason was quickly clear: the trees used by the XX compiler typically have a dozen or more namespaces in scope, and the saving achieved by making a virtual copy of a subtree was swamped by the cost of coping with two different namespace contexts for the two logical trees sharing physical storage.

In fact, it appeared that the costs of copying subtrees in this application had very little to do with the copying of elements and attributes, and were entirely dominated by the problem of copying namespaces.

We then experimented by using the "bulk copy" implementation instead of the virtual tree grafting. This gave a small but useful performance benefit (around 50ms, say 4%).

We considered various ways to reduce the overhead of namespace copying. One approach is to try and reduce the number of namespaces declared in the stylesheet that we are compiling; but that's cheating, it changes the input of the task we're trying to measure. Unfortunately the semantics of the XSLT language are very demanding in this area. Most of the namespaces declared in a stylesheet are purely for local use (for use in the names of functions and types, or even for marking up inline documentation), but the language specification requires that all these names are retained in the static context of every XPath expression, for use by a few rarely encountered constructs like casting strings to QNames, where the result depends on the namespaces in the source stylesheet. This means that the namespace declarations need to be copied all the way through to the generated SEF file. Using `exclude-result-prefixes` does not help: it removes the namespaces from elements in the result tree, but not from the run-time evaluation context.

We concluded there was little we could do about the cost of copying, other than to try to change the XSLT code to do less of it. We've got ideas about changes to the TinyTree representation of namespaces that might help[14], but that's out of scope for this project.

Recognizing that the vast majority of components (templates, functions, etc) contain no internal namespace declarations, we introduced an early check during

[14]See blog article: http://dev.saxonica.com/blog/mike/2019/02/representing-namespaces-in-xdm-tree-models.html

the static phase so that such components are labelled with an attribute `uniformNS="true"`. When this attribute is present, subsequent copy operations on elements within the component can use `copy-namespaces="false"` to reduce the cost.

Meanwhile, our study of what was going on internally in Saxon for this transformation yielded a few further insights:

- We found an inefficiency in the way tunnel parameters were being passed (this stylesheet uses tunnel parameters very extensively).
- We found some costs could be avoided by removing an `xsl:strip-space` declaration.
- We found that `xsl:try` was incurring a cost invoking `Executors.newFixedThreadPool()`, just in case any multithreading activity started within the scope of the `xsl:try` needed to be subsequently recovered. Solved this by doing it lazily only in the event that multi-threaded activity occurs.
- We found that during a copy operation, if the source tree has line number information, the line numbers are copied to the destination. Copying the line numbers is inexpensive, but the associated module URI is also copied, and this by default walks the ancestor axis to the root of the tree. This copying operation seems to achieve nothing very useful, so we dropped it.

At this stage, we were down to 825ms.

4.4.4. Algorithmic Improvements

In two areas we developed improvements in data representation and associated algorithms that are worth recording.

Firstly, import precedence.

All the components declared in one stylesheet module have the same import precedence. The order of precedence is that a module has higher precedence than its children, and children have higher precedence than their preceding siblings. The precedence order can be simply computed in a post-order traversal of the import tree. The problem is that annotating nodes during a post-order traversal is expensive: attributes appear on the start-tag, so they have to be written before writing the children. The existing code was doing multiple copy operations of entire stylesheet modules to solve this problem, and the number of copy operations increased with stylesheet depth.

The other problem here is that passing information back from called templates (other than the result tree being generated) is tedious. It's possible, using maps, but generally it's best if you can avoid it. So we want to allocate a precedence to an importing module without knowing how many modules it (transitively) imported.

The algorithm we devised is as follows. First, the simple version that ignores `xsl:include` declarations.

- We'll illustrate the algorithm with an alphabet that runs from A to Z. This would limit the number of imports to 26, so we actually use a much larger alphabet, but A to Z makes things clearer for English readers.
- Label the root stylesheet module Z
- Label its `xsl:import` children, in order, ZZ, ZY, ZX, ...
- Similarly, if a module is labelled PPP, then its `xsl:import` children are labelled, PPPZ, PPPY, PPPX, ...
- The alphabetic ordering of labels is now the import precedence order, highest precedence first.

A slight refinement of the algorithm is needed to handle `xsl:include`. Modules are given a label that reflects their position in the hierarchy taking both `xsl:include` and `xsl:import` into account, plus a secondary label that is changed only by an `xsl:include`, not by an `xsl:import`.

Secondly, types.

We devised a compact string representation of the XPath `SequenceType` construct, designed to minimize the cost of parsing, and capture as much information as possible in compact form. This isn't straightforward, because the more complex (and less common) types, such as function types, require a fully recursive syntax. The representation we chose comprises:

- A single character for the occurrence indicator (such as "?", "*", "+"), always present (use "1" for exactly one, "0" for exactly zero)
- A so-called alphacode for the "principal" type, chosen so that if (and only if) T is a subtype of U, the alphacode of U is a prefix of the alphacode of T. The alphacode for `item()` is a zero-length string; then, for example:
  - `N = node()`
  - `NE = element()`
  - `NA = attribute()`
  - `A = xs:anyAtomicType`
  - `AS = xs:string`
  - `AB = xs:boolean`
  - `AD = xs:decimal`
  - `ADI = xs:integer`
  - `ADIP = xs:positiveInteger`
  - `F = function()`
  - `FM = map()`

and so on.

- Additional properties of the type (for example, the key type and value type for a map, or the node name for element and attribute nodes) are represented by a compact keyword/value notation in the rest of the string.

Functions are provided to convert between this string representation and a map-based representation that makes the individual properties directly accessible. The parsing function is a memo function, so that conversion of commonly used types like "1AS" (a single `xs:string`) to the corresponding map are simply lookups in a hash table.

This representation has the advantage that subtype relationships between two types can in most cases be very quickly established using the `starts-with()` function.

It might be noted that both the data representations described in this section use compact string-based representations of complex data structures. If you're going to hold data in XDM attribute nodes, it needs to be expressible as a string, so getting inventive with strings is the name of the game.

4.4.5. Epilogue

With all the above changes, and a few others not mentioned, we got the elapsed time for the transformation down to 725ms, within a whisker of the target.

It then occurred to us that we hadn't yet used any of Saxon's multi-threading capability. We found a critical `xsl:for-each` at the point where we start XPath processing for each stylesheet component, and added the attribute `saxon:threads="8"`, so effectively XPath parsing of multiple expressions happens in parallel. This brings the run-time down to 558ms, a significant over-achievement beyond the target. It's a notable success-story for use of declarative languages that we can get such a boost from parallel processing just by setting one simple switch in the right place.

4.5. So what about Javascript?

The raison-d'etre of this project is to have an XSLT compiler running efficiently on Node.js; the astute reader will have noticed that so far, all our efforts have been on Java. Phase 2 of this project is about getting the execution time on Javascript down, and numerically this is a much bigger challenge.

Having achieved the Java numbers, we decided we should test the "tuned up" version of the compiler more thoroughly before doing any performance work (there's no point in it running fast if it doesn't work correctly). During the performance work, most of the testing was simply to check that the compiler could compile itself. Unfortunately, as already noted, the compiler only uses a fairly small subset of the XSLT language, so when we went back to running the full test

suite, we found that quite a lot was broken, and it took several weeks to repair the damage. Fortunately this did not appear to negate any of the performance improvements.

Both Node.js and the Chrome browser offer excellent profiling tools for Javascript code, and we have used these to obtain initial data helping us to understand the performance of this particular transformation under Saxon-JS. The quantitative results are not very meaningful because they were obtained against a version of the XX compiler that is not delivering correct results, but they are enough to give us a broad feel of where work is needed.

Our first profiling results showed up very clearly that the major bottleneck in running the XX compiler on Saxon-JS was the XPath parser, and we decided on this basis to rewrite this component of the system. The original parser had two main components: the parser itself, generated using Gunther Rademacher's REx toolkit [reference], and a back-end, which took events from the parser and generated the SEF tree representation of the expression tree. One option would have been to replace the back-end only (which was written with very little thought for efficiency, more as a proof of concept), but we decided instead to write a complete new parser from scratch, essentially as a direct transcription of the Java-based XPath parser present in Saxon.

At the time of writing this new parser is passing its first test cases. Early indications are that it is processing a medium sized XPath expression (around 80 characters) in about 0.8ms, compared with 8ms for the old parser. The figures are not directly comparable because the new parser is not yet doing full type checking. Recall however that the XX compiler contains around 5000 XPath expressions, and if the compiler is ever to run in 1s with half the time spent doing XPath parsing, then the budget is for an average compile time of around 0.1ms per path expression. We know that most of the path expressions are much simpler than this example, so we may not be too far out from this target.

Other than XPath parsing, we know from the measurements that we available that there are a number of key areas where Saxon-JS performance needs to be addressed to be competitive with its Java cousin.

- Pattern matching. Saxon-JS finds the template rule for matching a node (selected using `xsl:apply-templates`) by a sequential search of all the template rules in a mode. We have worked to try and make the actual pattern matching logic as efficient as possible, but we need a strategy that avoids matching every node against every pattern. Saxon-HE on Java builds a decision tree in which only those patterns capable of matching a particular node kind and node name are considered; Saxon-EE goes beyond this and looks for commonality across patterns such as a common parent element or a common predicate. We've got many ideas on how to do this, but a first step would be to replicate the Saxon-HE design, which has served us well for many years.

- Tree construction. Saxon-JS currently does all tree construction using the DOM model, which imposes considerable constraints. More particularly, Saxon-JS does all expression evaluation in a classic bottom-up (pull-mode) fashion: the operands of an expression are evaluated first, and then combined to form the result of the parent expression. This is a very expensive way to do tree construction because it means repeated copying of child nodes as they are added to their parents. We have taken some simple steps to mitigate this in Saxon-JS but we know that more needs to be done. One approach is to follow the Saxon/J product and introduce push-mode evaluation for tree construction expressions, in which a parent instructions effectively sends a start element event to the tree builder, then invokes its child instructions to do the same, then follows up with an end element event. Another approach might be to keep bottom-up construction, but using a lightweight tree representation with no parent pointers (and therefore zero-overhead node copying) until a subtree needs to be stored in a variable, and which point it can be translated into a DOM representation.
- Tree navigation. Again, Saxon-JS currently uses the DOM model, which has some serious inefficiencies built in. The worst is that all searching for nodes by name requires full string comparison against both the local name and the namespace URI. Depending on the DOM implementation, determining the namespace URI of a node can itself be a tortuous process. One way forward might be to use something akin to the Domino model recently introduced for Saxon-EE, where we take a third party DOM *as is*, and index it for fast retrieval. But this has a significant memory footprint. Perhaps we should simply implement Saxon's TinyTree model, which has proved very successful.

All of these areas impact on the performance of the compiler just as much as on the performance of user-written XSLT code. That's the dogfood argument for writing a compiler in its own language: the things you need to do to improve run-time performance are the same things you need to do to improve compiler performance.

5. Conclusions

Firstly, We believe we have shown that implementing an XSLT compiler in XSLT is viable.

Secondly, we have tried to illustrate some of the tools and techniques that can be used in an XSLT performance improvement exercise. We have used these techniques to achieve the performance targets that we set ourselves, and we believe that others can do the same.

The exercise has shown that the problem of the copying overhead when executing complex XSLT transformations is real, and we have not found good answers. Our previous attempts to solve this using virtual trees proved ineffec-

tive because of the amount of context carried by XML namespaces. We will attempt to make progress in this area by finding novel ways of representing the namespace context.

Specific to the delivery of a high-performing implementation of Saxon on the Javascript platform, and in particular server-side on Node.js, we have an understanding of the work that needs to be done, and we have every reason to believe that the same techniques we have successfully developed on the Java platform will deliver results for Javascript users.

References

[1] John Lumley, Debbie Lockett, and Michael Kay. February, 2017. XMLPrague. *XPath 3.1 in the Browser*. 2017. `http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf`

[2] John Lumley, Debbie Lockett, and Michael Kay. August, 2017. Balisage: The Markup Conference. *Compiling XSLT3, in the browser, in itself*. 2017. `https://doi.org/10.4242/BalisageVol19.Lumley01`

[3] Michael Kay. August, 2007. Extreme Markup. Montreal, Canada. *Writing an XSLT Optimizer in XSLT*. 2007. `http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html`

[4] Michael Kay. February, 2018. XML Prague. Prague, Czechia. *XML Tree Models for Efficient Copy Operations*. 2018. `http://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf`

XProc in XSLT: Why and Why Not

Liam Quin
Delightful Computing
<liam@fromoldbooks.org>

Abstract

Version 3 of XSLT includes the ability to call a secondary XSLT transformation from an XPath expression, using fn:transform(). It's also possible in most XSLT implementations to call out to system functions, so that one can arrange to run an arbitrary external program from within a stylesheet. The abilities from EXpath to read and write files, process zip archives, create and remove files and so forth also exist.

This integration means that it's often easier to call fn:transform() from a wrapper stylesheet than to use an XProc pipeline. Doing this, however, brings both advantages and disadvantages. This paper expands on the technique in more detail, illustrates some things that cannot be achieved, together with techniques to mitigate the limitations. The conclusion incorporates suggestions on when it is better to move to XProc or other tools.

Keywords: XSLT, XProc, pipelines

1. Introduction

Recently the author was working with a client whose staff were familiar with XSLT [3] but not with XProc [1] or XQuery [2]; it's not that they were unwilling to learn but that the ability to get the job done with the tools they already used was a benefit. In the process of solving their needs using XSLT, the author found that other consultants were using the same or very similar techniques.

As an experiment, the author rewrote an example XProc pipeline by Alex Miłowski to create an EPUB 3 electronic book; using XSLT the result was smaller and seemed easier to understand. However, getting to that result required some care and expertise, so the motivation for this paper in part is to share that expertise. Since there is ongoing work on XProc, another motivation is to ask some questions about possible simplifications to XProc that may make it easier for people to adopt it in the future.

A motivation for the work itself was that the primary implementation of XProc to which I had access at the time suffered from poor diagnostic messages, greatly increasing the cost of pipeline development. One approach would be to try and improve the messages, since the implementation is open source, but there was no guarantee that the changes would be accepted and in the short term I had

a client who needed an XSLT-only solution, so it wasn't until the work was done that the parallel with XProc became clearer. However, the possibility of an XProc checker in XSLT seemed worth exploring and may be a secondary result of the work.

2. The overall approach taken

The simplest approach to solving the author's clients' problems was to write a single XSLT stylesheet that handled each problem, and that was what was requested and done. But in one case the actual problem included an XML document describing steps to be taken. Interpreting that input one step at a time was actually adequate, but is what gave rise to the question about XProc. So the approach that the author tested was to hand-craft pipelines using XSLT, but to spend the time doing small experiments and finding ways to do more.

It is easy to see that a two-art pipeline such as the following:

```
XSLT to assemble input | XSLT to make HTML
```

could be written as

```
<xsl:value-of select="make-html(assemble-input())"/>
```

The two functions called can each (for example) call fn:transform() to invoke an external stylesheet.

Using function calls to replace sequential steps works well in simple cases. However, consider a pipeline with multiple connections such as that of Figure 1:

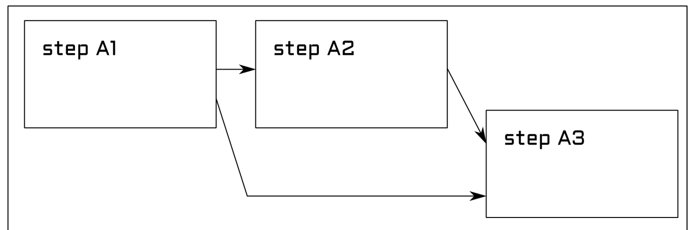


Figure 1. A pipeline in which the first step, A1, makes two outputs: one goes through step A2 to get to the final step, A3, and the other goes directly to A3 as its other input.

We cannot rewrite the pipeline in Figure 1 with the equivalence that A followed by B is the same as B(A()). If we try, we end up with:

```
A3(A2(A1()), A1())
```

Although this is correct in theory, it's not a very good theory, because it fails to take side-effects into account. If step A1() has side-effects, such as renaming files, calling it twice might duplicate them. The nature of XSLT is that a processor can cache the result of functions with the same arguments, so A1() might only be evaluated once, but this uncertainty is not acceptable in an XProc context.

One solution is to use temporary variables:

```
<xsl:variable name="tmp1" select="A1()"/>
    <xsl:sequence select="A3(A2($tmp1), $tmp1)"/>
```

The use of temporary variables poses other problems:

- There are restrictions on creating external resources using xsl:result-document when a variable is being constructed. This can be mitigated using fn:transform(), as described below.
- Memory usage may be higher, especially when compared to a text-based pipeline, because the result of A1() now cannot be consumed in streaming mode. However, actual pipeline implementations generally pass memory-based trees from one processor to the next when they can, so this may not be a difference in practice.
- A pipeline processor, and to some extent an XSLT processor using functions but without temporary variables, may be able to make better use of multiple threads. However, this depends on the optimizer of the specific XSLT processor used, and optimization technology is sophisticated.

Fortunately, the restriction on creating external resources does not apply when using a variable to store the result of fn:transform() calling an external transform that creates such resources. Instead, the resources are not created, but fn:transform() returns a map containing them, and this map can be associated with a variable.

3. Immediate or Compiled

The author's first attempt to gain pipeline functionality in XSLT through functional composition used a compilation technique: an XSLT stylesheet that read a simple pipeline specification and wrote out a stylesheet that, when interpreted by an XSLT processor, would give the desired result. This worked, but the existence of fn:transform() made the author wonder whether it would be possible to interpret the input document directly. It was indeed possible. Although there were some difficulties along the way, the result was easier to use, more reliable, and faster than the two-step approach.

4. XProc Features

There is no attempt here to write a complete XProc implementation in XSLT, although it's in principle possible. Rather, this paper compares writing comparable processing in XSLT and XProc with a strong focus on the XSLT side. The reason is that if writing XProc pipelines was not a goal of the author's clients, then while solving their problems directly in XSLT would help them, writing an XProc

implementation would not be work in which they would be interested in investing.

4.1. Definition

Any attempt to compare writing pipelines in XSLT to XProc needs first to explain what is meant by a pipeline and secondly to list the features of XProc that the author considers relevant.

The term *pipeline* (or pipe, informally) refers to a sequence or non-acyclic graph of steps, each of which uses as input the output of one or more preceding steps.

This definition is not that used in the XProc specification exactly, and should be considered loose, approximate, general: as the paper progresses the intent of the term should become clearer, but the most important thing is that we may speak of a pipeline without necessarily meaning one represented in the XProc language.

4.2. Comparing some features

The XML Pipeline language has some features (some of which come through extensions and are not in the original specification) that

4.3. Dependency Management

Although an obvious form of dependency in a pipeline could be expressed as *uses as input the result of*, a less obvious one might be *resource has a less recent change-date than*. This sort of dependency is used by utilities such as *make* and *ant* that are primarily intended for the development of computer programs: “if the source file has changed since the program was last compiled, recompile the source file to rebuild the program”.

The Calabash XProc implementation supports file revision-time dependencies; in XSLT we can emulate them using the EXPath `file:date()` function:

```
<rule produces="socks.html" depends="socks.xml socks.xslt">
  <xslt input="socks.xml" output="socks.html" stylesheet="socks.xslt"/>
</rule>
```

An XSLT fragment might look like,

```
<xsl:if test="file:date('socks.xml') gt file:date('socks.html')">
  <xsl:sequence select="fn:transform(...)"
</xsl:if>
```

We will expand on the incomplete expression later, after discussing files, resources, and reading our own output, a significant consideration in any XSLT-based approach.

It is worth noting that the use of external parsed entities, of XInclude, DITA-style maps and other multi-resource machinations can mean that determining that an XML document has been modified in general is difficult. However, explicit rules listing all of the fragments can mitigate this, as can making one's XSLT understand the inclusion paradigm being used. and this difficulty is not of course specific to any particular implementation approach.

4.4. Running external processes

It's possible to run arbitrary external code in most XSLT processors, including Saxon. The exact way to do this varies depending on the host programming language and the particular XSLT engine. It usually involves making an XPath function available which, when run, will cause execution of an external process. The main difficulties are ensuring that the XSLT engine doesn't optimize out the call to the external program and capturing the output. One common approach to prevent optimization is to include the "result" of calling the extension function (usually the empty sequence or a status code) in an `xsl:message` or in a result element that's used as input to the next stage. For example, it's possible to write out a Perl script with `xsl:result-document` and then to run it; the XSLT in Appendix 1 does this. However, see the section on reusing persisted resources later in this paper for some important caveats.

Capturing the output of the command generally involves writing it to a temporary file and then reading that file, because some environments make it hard to distinguish between error messages and output, or ignore the output completely unless you set up some sort of pipe mechanism. For production purposes it's important to ensure that error messages are captured and handled.

4.5. Expression Languages and Variables

The XProc language supports the use of variables whose values can be associated with items fished out of the pipeline stream using XPath expressions. With a two-phase XSLT compilation it was possible to copy XPath expressions into the generated XSLT stylesheet. Without that, it's possible to use eval() in XSLT processors that support it; it would also be possible to parse the XPath expressions in XSLT, but that seems unnecessarily burdensome. For now, the author chose not to support variables, although simple variables could certainly be handled.

Note that variables are always evaluated in the context of the pipeline so far. With a functional approach, this means passing them up from fn:transform() to the calling function as part of the result map and, potentially, down to called transformations as a parameter. Recall that XProc variables are restricted to string values (or, more precisely, for XPath 2, anyUntypedAtomic) and hence can be represented completely as strings.

4.6. Reading and Writing Archives

The EXpath archive module provides the ability to read and write Zip archives. Zip here is not the same as gzip despite the similar name, but *is* the format used by EPUB for ebooks. As a result, XSLT stylesheets can now read and write ebooks. A caveat is that the first entry in an EPUB archive must not be compressed; the most portable way to achieve this is to make a zip archive containing only the one constant mediatype file that is required; read this archive in XSLT (or include it in the stylesheet in a Base64 string variable) and then use the archive functions to append to that archive, but with compression.

5. Limitations and Restrictions of the XSLT Approach

It may seem from the foregoing that there's no reason to use XProc in an environment in which XSLT is in heavy use. However, there are areas in which the difficulty of orchestrating XSLT transforms outweighs the advantage of using a single language.

5.1. Restrictions on Reading Created Resources

From [3] we can see (section 25.2) restrictions on the use of xsl:result-document; some of these do not concern us here, but pay particular attention to the following:

- It is a dynamic error to evaluate the xsl:result-document instruction in temporary output state.
- It is a dynamic error for a transformation to generate two or more final result trees with the same URI.
- It is a dynamic error for a stylesheet to write to an external resource and read from the same resource during a single transformation, if the same absolute URI is used to access the resource in both cases.

The first of these restrictions means you can't write to external files (resources) while you are constructing a variable. Since the strategy for implementing complex pipelines involves retaining the output from earlier steps in variables, this means that steps cannot write to external resources. Fortunately, none of these restrictions applies to fn:transform(): you can write to an external file in a stylesheet you call, and then read from it in the caller, as long as you ensure that there is a dependency so that the XSLT engine evaluates the external stylesheet before you try to read the result.

In fact, when you use fn:transform(), external resources are not written out. Instead, fn:transform() either returns a map containing them, or calls a function you supply once for each such resource. The *called* stylesheet processor is not in a temporary output state.

It is possible to evade the restrictions on xsl:result-document using file:write() and file:read(), and by not attempting to read a file that you know may have changed except by calling an external stylesheet to do so. This last is not onerous as you can construct a minimal stylesheet on the fly and pass it to fn:transform() as a stylesheet node, but you need to be aware that the XSLT processor is likely to cache the result of file:open() on any given URI.

Note that when an XSLT processor is embedded in another application, such as a beautiful and highly functional XML editor, the result of fn:transform() may be filtered by that application's entity or resource manager, so that it turns out to be wise to handle both return mechanisms: provide a function and *also* check the returned map.

The following simple stylesheet writes a file and then reads it back again.

```
<xsl:stylesheet version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:file="http://expath.org/ns/file"
  exclude-result-prefixes="#all">

  <xsl:output method="text" />

  <xsl:variable name="socks" as="element(socks)">
    <socks>black</socks>
  </xsl:variable>

  <xsl:template match="/">
    <!--* write the file *-->
    <xsl:value-of
      select='file:write("file:/tmp/socks.xml", $socks)' />
    <!--* Read it back, upper case to show we did it: *-->
    <xsl:value-of
      select='upper-case(doc("file:/tmp/socks.xml")//socks/text())' />
  </xsl:template>
</xsl:stylesheet>
```

In the example above, file:write() is used and avoids the restrictions on xsl:result-document. The output is simply the word “black” but transformed into upper case. Inspection of the file created will show the word in lower case, by which we know that the output from the stylesheet is from reading the document.

When you use fn:transform to call a subsidiary stylesheet, any resources that the subsidiary stylesheet tries to create using xsl:result-document are returned in a map; in addition, you can arrange for a user-defined function to be called for each such resource. The following example illustrates that:

```
<xsl:stylesheet version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:file="http://expath.org/ns/file"
```

```
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:delicomp="https://www.dlightfulcomputing.com/"
  expand-text="yes">

  <xsl:output method="text" />

  <xsl:template match="/">
    <xsl:variable name="man" select="transform(
        map {
          'stylesheet-location' : 'output-two.xsl',
          'source-node' : /,
          'delivery-format' : 'serialized',
          'post-process' : delicomp:do-output#2,
          'stylesheet-params' : map {
            QName('', 'colour') : 'heavy iron',
            QName('', 'garment') : 'collar'
          }
        }
      )"/>
    <xsl:message>Reading from file: {
        doc(map:keys($man)[1])
    }</xsl:message>
  </xsl:template>

  <xsl:function name="delicomp:do-output">
    <xsl:param name="uri" />
    <xsl:param name="value" />
    <xsl:message>write to {$uri}</xsl:message>
    <xsl:value-of select="file:write-text($uri, $value)" />
  </xsl:function>
</xsl:stylesheet>
```

The subsidiary stylesheet here is as follows:

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform"
  xmlns:output="http://www.w3.org/2010/xslt-xquery-serialization">
  <xsl:param name="garment" select=" 'sock' " />
  <xsl:param name="colour" select=" 'green' " />

  <xsl:template match="/">
    <xsl:result-document href="inner-garment.xml">
      <garment>
        <xsl:value-of select="concat('Simon wears a ', $colour, ' ',
$garment, '&#xa;')"/>
      </garment>
    </xsl:result-document>
```

```
  </xsl:template>
</xsl:stylesheet>
```

Running the main stylesheet with itself as input produces this output:

```
write to file:inner-garment.xml
Reading from file: Simon wears a heavy iron collar
```

Noteworthy in this example is that if the call to xsl:message that uses $man is removed, the file is not created (depending on the XSLT processor used). To get the side-effect, the result must be used in some way.

If the secondary stylesheet had in turn called a tertiary stylesheet in the same way, it would need to save the results.

If a stylesheet creates a file but does not need to read it back (the usual case) it can simply use xsl:result-document directly; if it is called by another stylesheet, that stylesheet will have to save the output files or they will be lost. As a result, normally, every call to fn:transform() that might produce external resources must be accompanied by appropriate file creation.

5.2. Other Languages

This section would be incomplete if it did not mention that there are no standard ways to call XProc or XQuery from XSLT. Of course, if you are using XSLT to avoid working with XProc, the first of these will not be an issue. At least one XSLT implementation (Saxon) provides a query() function analogous to transform(), but in the use case of orchestrating multiple XQuery fragment, XProc (including the existing XProc implementation in XQuery, perhaps) or even using fn:transform and file:write from XQuery, makes sense.

5.3. Validation, XInclude, and Other Parsing options

Although a particular XSLT implementation might offer XInclude support, it is not part of XSLT: there is no two-argument variant of fn:document() that takes options, for example. XInclude can be implemented in XSLT, and has been, as well as in XSLT processors; control over DTD validation such as that provided by XProc's p:load step is absent, however, and requires use of extensions.

6. Conclusions

Today, XSLT 3 (when used with widely-implemented extensions such as the file:write() function) can emulate much of the functionality of XProc. It is entirely reasonable to use XSLT to orchestrate simple pipelines, and doing so in an environment or business culture in which XSLT is the main language used for XML processing can help to minimize development and maintenance costs.

There are difficulties in handling relationships between files in XSLT that can be mostly worked around, but the result in some cases may be no easier to maintain than an XProc pipeline.

In environments where the pipelines are complex, or where multiple languages are used to process XML documents, XProc remains a valuable tool. In addition, for large documents, interactions with the streaming mode of XSLT may be easier to manage with XProc.

The conclusion, then, is that fn:transform, file:read, file:write, and other functions go a long way towards reducing the need for XProc in XSLT environments but are not an easy drop-in replacement. Pipelines still have a valuable role to play.

Bibliography

[1] Miłowski, Alex; Thompson, Henry S.; Walsh, Norman (Eds.) *XXProc: An XML Pipeline Language*. Recommendation, May 2010, W3C. `http://www.w3.org/TR/2010/REC-xproc-20100511/`

[2] Dyck, Michael; Robie, Jonathan; Spiegal, Josh (Eds.) *XQuery 3.1: An XML Query Language*. Recommendation, March 2017, W3C. `https://www.w3.org/TR/xquery-31/`

[3] Kay, Michael (Ed.) *XSL Transformations (XSLT) Version 3.0* . Recommendation, June 2017, W3C. `https://www.w3.org/TR/2017/REC-xslt-30-20170608/`

Merging The Swedish Code of Statutes (SFS)

Ari Nordström
<ari.nordstrom@gmail.com>

Abstract

In 2017, Karnov Group, a Danish legal publisher, bought Norstedts juridik, a Swedish legal publisher, and set about to merge their document sources. Both companies publish the Swedish legislation, known as the Swedish Code of Statutes or SFS, online and in print, and both use in-house XML sources based on PDF sources provided by Regeringskansliet, i.e. the Swedish government.

But the companies use separate tag sets and their own interpretations of the semantics in the PDFs. They also enrich the basic law text with extensive annotations, links to (and from) caselaw, and so on. Norstedts also publishes the so-called blue book, a yearly printed book of the entire in-force SFS.

It doesn't make sense to continue maintaining the two SFS sources separately, of course. Instead, we want, essentially, the sum of the two, with annotations, missing versions (significant gaps exist in the version histories), etc added.

This paper is about merging the SFS content into a single XML source. Basically, we convert both sources into a single exchange format, compare those using Delta XML's XML Compare, and then do the actual merge based on the diff. Finally, we convert the merged content to a future editing format.

1. Intro & Background

1.1. The Swedish Code of Statutes

From Wikipedia[1]:

> *The Swedish Code of Statutes (Swedish: Svensk författningssamling; SFS) is the official law code of Sweden which contains the statutes and ordinances enacted and designated by the Government, including a publication of all new Swedish laws enacted by the Riksdag.*

The Swedish law, as is the case with most sets of statutes regardless of country, is constantly changing, with older paragraphs, chapters, or indeed entire laws being

repealed in favour of new ones. Today, these older versions are saved to preserve the evolution of the law to the benefit of the lawyers that use them, marking up the sources with valid-from and valid-to dates, as well as the ID of the law, known as the *change SFS number*, used to repeal the old versions. This wasn't always the case, however; before the advent of the Internet, older versions of the law would only be available in printed form. If you were a lawyer and needed to know when something changed, the printed sources were your best bet.

This changed in the 90s when legal publishers wanted to make the law available on CD-ROMs and the Internet, and began to save the repealed versions as well as add the missing ones. With Sweden joining the EU during that same period, it was impossible to recreate everything, of course, so they would add old versions to some of the more important laws. Not every version and certainly not every law, but the more recent and the more important ones.

1.2. Merging Companies (and Content)

In 2017, *Karnov Group*, a Danish legal publisher and the employer of yours truly, bought *Norstedts juridik*, a Swedish legal publisher, and set about to merge not only the two companies but the information published by them. As legal publishers, both companies publish the Swedish Code of Statutes online and in print. Both do it from in-house XML sources, and both sources are based on the same basic text of the law, provided by *Regeringskansliet*, i.e. the Swedish government, usually as PDFs.

But the companies use separate tag sets to mark up the contents, and, of course, their own interpretations of the semantics in the PDFs. They also enrich the basic law text with extensive annotations and editorial commentary, links to (and from) caselaw, and so on. And, in the case of Norstedts, the XML source is continually updated to support their flagship product, the so-called blue book, a printed book of the entire in-force SFS, updated yearly as the law itself changes.

When publishers added CD-ROMs and other electronic formats to their offerings in the 90s, both companies set out to add older SFS content in addition to the ever-changing new legislation. As indicated above, however, there was only time and means to add the more “important” ones, and so there would be significant gaps in the respective version histories of the laws at both Karnov and Norstedts, and neither company's sources would match the other's entirely.

1.3. Merging SFS Content

Enter early 2018 and yours truly starting a new job as a Senior XML Geek at Karnov Group. At the time, the work had started to merge the two companies and their information, and it fell upon me to suggest, and implement, a course of action for SFS.

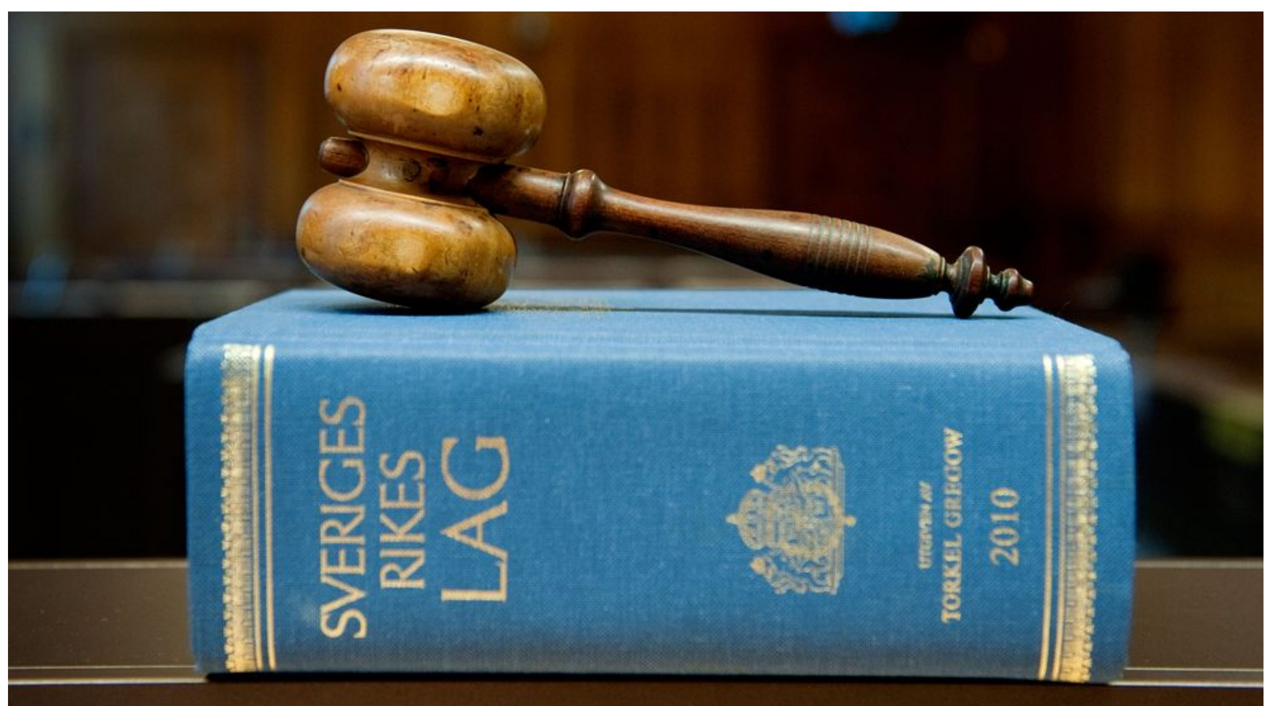


Figure 1. The Printed Law Book

Ideally, of course, best is to maintain a single source for the entire set of Swedish statutes. At first glance, then, it might be enough to simply pick one XML source and use that.

However, both companies had their existing online systems and customer bases, with differing product offerings. As suggested above, the sources described the same thing — the Swedish Code of Statutes — but the sets were not an exact match. Sometimes one company would have SFS documents the other didn't, and often, there were significant gaps with older versions of individual chapters and paragraphs. It made little sense to throw any of that away just to make the merge simpler.

There was also the matter of the Norstedts flagship product, the printed law book, that *had to* be included in any future offerings, so no matter what, we'd need to include anything written specifically for that book.

Similarly, Karnov included extended notes in their SFS content and made that available as a separate product, which meant that they would have to be preserved, too. Obviously, in one way or another, the respective SFS sources would have to be merged.

2. The Merge Process

This section briefly describes the basic merge process. Here, and in the rest of the paper, I'll use the abbreviations “KG” and “NJ” for *Karnov Group* and *Norstedts juridik*, respectively.

2.1. What You Need to Understand First

First of all, let me just emphasise how the Karnov Group (KG) and Norstedts juridik (NJ) XML sources come into being. In most cases, the editors receive PDFs from the government. These days, the PDFs are the output of a word processor[1], but not so very long ago, they could just as well be scans. These are then copied and pasted into *whatever tags and structures that the editor deems to fit best*. This is usually based on the formatting — font sizes, margins, and so on — of the content as presented in the PDFs. For a law paragraph, the group of text blocks that is the basic unit of most laws, this is usually straight-forward, but for any grouping of content, from sections to chapters to parts, this is often a matter of interpretation based on formatting and the perceived legal use.

In other words, *there are no absolutes here!*

2.2. Assumptions and Approach

My initial assumption was that for any shared SFS document — a law present in both sources — the basic text of the law itself would be the same albeit with different tags and semantics, and that it would therefore be possible to recognise any additions — annotations, comments, links, and other enrichments — as such, and allow us to keep those separate from the main content.

Obviously, a primary goal of the project was to unify the sources — a single source for the law text, with any additions intended for the various outputs neatly separated from the law text. Since the sources were quite different from each other, they first

My approach to merging the sources was basically as follows:

1. Create a DTD[2] describing a common *exchange XML format (EXC DTD)*, essentially the sum of the semantics found in the respective sources.
2. Update an existing, or create a new, *authoring format (KG++ DTD)*, in which the merged SFS corpus can be maintained and updated in the future.
3. Convert both sources to the common exchange format, at times up-converting and fixing various semantic constructs so they'd match each other.
4. Compare the converted sources with each other using an XML diffing tool that merges the sources and adds diff markup to indicate any differences.
5. Address those differences, one by one, to produce properly unified SFS documents with a single main law text and clearly identified enhancements, still in the exchange format.

[1]Guessing MS Word, in many cases.

[2]Both existing tool chains use DTDs so other schema languages were out of the question.

6. Convert the unified SFS documents to the authoring XML (KG++ DTD) format.

2.3. Legacy Publishing at NJ

The Norstedts publishing offerings, from the printed law book to legacy online publishing platforms, needed to be supported until the publishing *systems* had been properly merged to support the new content. While easily the subject of an entire paper by itself, from an SFS document merge point of view my approach was simple enough:

1. Once merged SFS content is updated in the new KG++ format, those updates are converted *back to the exchange format*.
2. The exchange documents are then (down-)converted to the old, legacy, NJ format since the legacy publishing processes at NJ all require that format.
3. The NJ legacy publishing processes can then *publish* the new content as if it had been produced in the old NJ format.

This, of course, can be done infinitely if needed. In practice, however, we'll only be doing it until there are new publishing processes in place to handle the NJ publishing offerings and the old systems can be retired[3].

This approach buys us time, basically.

2.4. Legacy Publishing at KG

While it is certainly possible to use the NJ legacy approach for KG legacy publishing (in other words, convert any updated KG++ content to EXC and, from there, back to the old KG format), the larger merge project is updating those systems as part of the project proper.

3. Implementation

The above gives an overview of what I set out to do. This chapter offers some detail on the implementation.

3.1. Transformation and Validation Pipelines

I'm an XProc[4] fan, having used it for anything from XML publishing to large conversion projects, and so an XProc pipeline to run the dozens of XSLT transformation steps I foresaw for each conversion (see Procedure) was my default strategy[4]. More specifically, I've successfully used Nic Gibson's *XProc Tools*[5] in the

[3]This needs to happen before the end of this year, for various non-technical reasons.

[4]I introduced XProc at Karnov, as a matter of fact.

past[8] to allow me to easily run an arbitrary number of XSLT stylesheets in sequence by listing them in a *manifest file*. Adding a step is a question of adding an `item` element pointing out the new XSLT stylesheet to the manifest.

Here's an entire manifest file for the pipeline converting EXC XML to KG++ XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="../../../../xproc-stuff/xproc-tools/schemas/
manifest.rng"
    type="application/xml" schematypens="http://relaxng.org/ns/structure/
1.0"?>
<manifest
    xmlns="http://www.corbas.co.uk/ns/transforms/data"
    xml:base="."
    description="This converts Exchange XML to KG++ format">

    <group description="Unmatched also need to be grouped">
        <item
            href="../sfsmerge/SFS-EXC-MERGE_group.xsl"
            description="Group unmatched; merged will not be touched"/>
        <item
            href="../sfsmerge/SFS-EXC-MERGE_no-level-group.xsl"
            description="Group group dividers without @level and their
following siblings (non-recursive)"/>
    </group>

    <group
        description="KG-specific steps"
        xml:base=".">
        <!-- Converts main structures -->
        <item
            href="SFS-EXC2KG_structure.xsl"
            description="Converts main EXC structures"/>

        <!-- Chapter versions to meta -->
        <item
            href="SFS-EXC2KG_version2meta.xsl"
            description="Converts EXC chapter version information to KG+
+ meta items"/>

        <!-- Group title versions if there are two or more consecutive
title groups -->
        <item
            href="SFS-EXC2KG_merge-title-versions.xsl"
            description="Merges the versions of two or more consecutive
title groups"/>
```

```
        <!-- Front matter -->
        <item
            href="SFS-EXC2KG_front.xsl"
            description="Converts front matter elements, except those
that only need a namespace conversion"/>

        <!-- In-force info -->
        <item
            href="SFS-EXC2KG_in-force.xsl"
            description="Converts in-force elements"/>

        <!-- Various body- and chapter-level elements -->
        <item
            href="SFS-EXC2KG_body.xsl"
            description="Converts various body- and chapter-level
elements"/>

        <!-- Para- and subpara-level elements -->
        <item
            href="SFS-EXC2KG_paras.xsl"
            description="Converts para- and subpara-level elements"/>

        <!-- Title elements -->
        <item
            href="SFS-EXC2KG_title-grp.xsl"
            description="Converts title elements"/>

        <!-- Block-level -->
        <item
            href="SFS-EXC2KG_block-level.xsl"
            description="Converts block-level elements"/>

        <!-- Annotations -->
        <item
            href="SFS-EXC2KG_annotations.xsl"
            description="Converts annotations (both redanm and
kommentar)"/>

        <!-- Inline elements -->
        <item
            href="SFS-EXC2KG_inline.xsl"
            description="Converts inline elements"/>

        <!-- EXC namespace to KG++ namespace -->
        <item
```

```
            href="SFS-EXC2KG_namespace-conversion.xsl"
            description="Converts the EXC namespace to the KG++
namespace"/>

        <!-- Convert various IDs to LOGIDs -->
        <item
            href="SFS-EXC2KG_id-logid.xsl"
            description="Converts IDs to LOGIDs"/>

        <!-- Attrs -->
        <item
            href="SFS-EXC2KG_attrs.xsl"
            description="Converts various attributes"/>
    </group>

</manifest>
```

I'll not go into detail on how Nic's tools work here (for a more through explanation, see [5] for the code — it's open source, so if you're interested, try it out! — and [8] for an example of their use); suffice to say that they allow me to write short and to the point XSLT stylesheets, each of them doing one thing and one thing only. A step can be as simple as this:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:exc="http://karnovgroup.com/ns/exchange"
    xmlns:kgp="http://ns.karnovgroup.com/kg-pp"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    exclude-result-prefixes="xs"
    version="2.0">

    <!-- This step transforms EXC inline elements to KG++ ditto -->

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="/">
        <xsl:apply-templates select="node()" mode="SFS-EXC2KG_INLINE"/>
    </xsl:template>

    <xsl:template match="exc:ref" mode="SFS-EXC2KG_INLINE">
        <kgp:ref>
            <xsl:copy-of select="@* except @type"/>
            <xsl:if test="@type">
                <xsl:attribute name="target-type" select="@type"/>
            </xsl:if>
            <xsl:apply-templates select="node()" mode="SFS-
```

```
EXC2KG_INLINE"/>
        </kgp:ref>
    </xsl:template>

    <!-- ID transform -->
    <xsl:template match="node()" mode="SFS-EXC2KG_INLINE">
        <xsl:copy copy-namespaces="no">
            <xsl:copy-of select="@*"/>
            <xsl:apply-templates select="node()" mode="SFS-
EXC2KG_INLINE"/>
        </xsl:copy>
    </xsl:template>

</xsl:stylesheet>
```

A huge difference for me as an XSLT developer is that I can focus on one issue or area at a time instead of having to write large and most likely modularised XSLTs that are hard to read and hard to debug. An XProc pipeline that runs a manifest file listing XSLT steps can produce debug output from each step, radically simplifying development:

```
8.2M Jan 18 11:34 0-SFS1962-0700.xml
8.2M Jan 18 11:34 1-SFS-EXC-MERGE_group.xsl.xml
8.2M Jan 18 11:34 2-SFS-EXC-MERGE_no-level-group.xsl.xml
8.2M Jan 22 09:17 3-SFS-EXC2KG_structure.xsl.xml
8.3M Jan 22 09:17 4-SFS-EXC2KG_version2meta.xsl.xml
8.2M Jan 22 09:17 5-SFS-EXC2KG_merge-title-versions.xsl.xml
8.3M Jan 22 09:17 6-SFS-EXC2KG_front.xsl.xml
8.2M Jan 22 09:17 7-SFS-EXC2KG_in-force.xsl.xml
8.3M Jan 22 09:17 8-SFS-EXC2KG_body.xsl.xml
8.3M Jan 22 09:17 9-SFS-EXC2KG_paras.xsl.xml
8.6M Jan 22 09:17 10-SFS-EXC2KG_title-grp.xsl.xml
8.8M Jan 22 09:17 11-SFS-EXC2KG_block-level.xsl.xml
8.3M Jan 22 09:17 12-SFS-EXC2KG_annotations.xsl.xml
9.8M Jan 22 09:17 13-SFS-EXC2KG_inline.xsl.xml
8.1M Jan 22 09:17 14-SFS-EXC2KG_namespace-conversion.xsl.xml
8.1M Jan 22 09:17 15-SFS-EXC2KG_id-logid.xsl.xml
7.9M Jan 22 09:17 16-SFS-EXC2KG_attrs.xsl.xml
```

The first file, prefixed “0-”, is simply a copy of the source file, here because it's needed by XSpec unit tests. The others are all outputs of the XSLT stylesheets they are named after (with the number prefixes there for sorting and general convenience), extremely useful when used as input to the following step when debugging that step.

I wrote XSpec unit tests for many, if not most[5], of my XSLT steps, but also an XProc pipeline that was able to run the XSpec unit tests in sequence following an

XSpec manifest file similar to the XSLT manifest file, using the debug output from the various steps as input wherever needed.

Each of the pipelines ended by validating the output against the DTD and, optionally, against a Schematron that made sure that the various features of the exchange format were being used correctly.

3.2. Examples

To give you an idea of the kind of differences between the two sources typically encountered, here is an example from KG:

```
<stycke chgbardate="20180701" logid="SFS1962-0700.K2.P2.S4" num="4">De
inskränkningar
av svensk domsrätt som anges i andra och tredje styckena gäller inte för
brott som avses i
    <list type="manuell">
        <listelement>1. <referens logidref="SFS1962-0700.K4.P1A">4 kap.
1 a</referens> och
            <referens logidref="SFS1962-0700.K4.P4C">4 c §§</referens>
och
            <referens logidref="SFS1962-0700.K16.P10A">16 kap. 10 a §
första stycket 1</referens> och
            <referens logidref="SFS1962-0700.K16.P10A.S5">femte stycket</
referens> eller
                försök till sådana brott,</listelement>
        <listelement>2. <referens logidref="SFS1962-0700.K4.P4.S2">4
kap. 4 § andra stycket</referens> varigenom
            någon förmåtts att ingå ett sådant äktenskap eller en sådan
äktenskapsliknande förbindelse som avses i
            <referens logidref="SFS1962-0700.K4.P4C">4 c §§</referens>
eller försök till sådant
            brott, eller</listelement>
        <listelement>3. <referens logidref="SFS1962-0700.K6.P1">6 kap. 1-
6</referens>,
            <referens logidref="SFS1962-0700.K6.P8">8</referens>,
            <referens logidref="SFS1962-0700.K6.P9">9</referens> och
            <referens logidref="SFS1962-0700.K6.P12">12 §§</referens>
eller försök till brott enligt
            <referens logidref="SFS1962-0700.K6.P1">6 kap. 1</referens>,
            <referens logidref="SFS1962-0700.K6.P2">2</referens>,
            <referens logidref="SFS1962-0700.K6.P4">4-6</referens>,
            <referens logidref="SFS1962-0700.K6.P8">8</referens>,
            <referens logidref="SFS1962-0700.K6.P9">9</referens> och
            <referens logidref="SFS1962-0700.K6.P12">12 §§</referens>,
```

[5] I freely admit that I should have written more of them.

```
om brottet begåtts mot
            en person som inte fyllt arton år.</listelement>
    </list>
</stycke>
```

Note the mixed content, allowing character data to be mixed with a block-level list. Also note that the list is manual, with list labels inserted by the author, and the `referens` elements marking up a citation.

Note

I've added line breaks to make the example easier to read. The KG sources were not pretty-printed.

Here's the NJ equivalent:

```
<sty>De inskränkningar av svensk domsrätt som anges i andra och tredje

    styckena gäller inte för brott som avses i</sty>

<lista typ="decimal">

    <lp><sty>4 kap. 1 a och 4 c §§ och 16 kap. 10 a § första stycket 1

        och femte stycket eller försök till sådana brott,</sty></lp>

    <lp><sty>4 kap. 4 § andra stycket varigenom någon förmåtts att ingå

        ett sådant äktenskap eller en sådan äktenskapsliknande
förbindelse

        som avses i 4 c § eller försök till sådant brott, eller</
sty></lp>

    <lp><sty>6 kap. 1-6, 8, 9 och 12 §§ eller försök till brott enligt

        6 kap. 1, 2, 4-6, 8, 9 och 12 §§, om brottet begåtts mot en
person

        som inte fyllt arton år. <andringssfs>

            <andringssfsd><sty>Lag (2018:618).</sty></andringssfsd>

            <andringssfstr>
                <sty>
                    <stil font-weight="bold">Lagar 2010:399</stil> (se
vid 35:4),
                    <stil font-weight="bold">2013:365, 2014:381</stil>
```

```
(se vid 4:4 c),
                     <stil font-weight="bold">2018:618,</stil> med
ikraftträdande

                 1 juli s.å.</sty>
            </andringssfstr>

        </andringssfs>
    </sty>
    </lp>

</lista>
```

Here, the introductory text to the list is separated into its own text paragraph (`sty`) element. The list is ordered (`typ="decimal"`) and the labels are inserted when publishing. There is no citation markup; the citations will be handled later by an auto-linking feature. There is also content not present in the KG version: the `andringssfs` provides information about what amended this particular law paragraph, with one structure for online and the other for print.

While I've added a few line breaks to the example to make it easier to read, the empty lines in the NJ text content, above, are present in the sources, adding a further complication to the diff[6].

3.3. KG/NJ to EXC

The two pipelines that convert KG and NJ XML to EXC contain most of the actual up-conversions and tag abuse fixes, as the respective sources in exchange format need to be reasonably aligned with each other in terms of various semantics so they can be successfully diffed and merged.

This may seem like a simple task, given that they both depict the same law texts, but in reality, there were lots of problems to address. The NJ to EXC pipeline currently lists 57 steps while the KG to EXC one adds a step to a total of 59 steps. A fair number of them address specific up-conversion or tag abuse problems (see Section 4.3) while others are needed to change the semantics in one source to better align with the other.

3.4. Diff and Merge

Once both sources are in exchange format, they need to be compared with each other. For this, we chose what many see as the industry standard for comparing XML these days, *Delta XML's XML Compare*[3] tool. XML Compare compares two XML files, “A” and “B”, with each other according to predefined rules[7] and inserts diffing markup to indicate where the differences lie:

[6]Why the extra lines? Your guess is as good as mine.

```
<exc:para deltaxml:deltaV2="A!=B">
    <deltaxml:attributes deltaxml:deltaV2="B">
        <dxa:kg-process
            deltaxml:deltaV2="B">
            <deltaxml:attributeValue deltaxml:deltaV2="B"
                >@logid="SFS2004-0046.K3.P3.S1" @num="1"
                @chgbardate="20161101"</deltaxml:attributeValue>
        </dxa:kg-process>
        <dxa:id
            deltaxml:deltaV2="B">
            <deltaxml:attributeValue deltaxml:deltaV2="B"
                >SFS2004-0046.K3.P3.S1</deltaxml:attributeValue>
        </dxa:id>
    </deltaxml:attributes>Fondbolaget
    och förvaringsinstitutet ska ingå ett skriftligt avtal som reglerar
förhållandet mellan
    parterna. Avtalet ska bland annat reglera det informationsutbyte och
den samordning som krävs
    för att institutet ska kunna utföra sina uppgifter för fondens
räkning i enlighet med kraven i
    denna lag och andra författningar.
    <deltaxml:textGroup deltaxml:deltaV2="A">
        <deltaxml:text
            deltaxml:deltaV2="A"> </deltaxml:text>
    </deltaxml:textGroup>
    <exc:ref deltaxml:deltaV2="B"
        source="kg" href="SFS2004-0046.N94" role="notreferens-KAR"
number="94"/>
    <exc:change-sfs-grp deltaxml:deltaV2="A" source="nj" role="2016:892">
        <exc:change-sfs publish-type="online">
            <exc:para>Lag (2016:892).</exc:para>
        </exc:change-sfs>
        <exc:change-sfs publish-type="print">
            <exc:para><exc:emph type="bold">Lag 2016:892</exc:emph> (se
vid 4:15).</exc:para>
        </exc:change-sfs>
    </exc:change-sfs-grp>
</exc:para>
```

Here, we have a diffed text paragraph where all attributes are "B" (KG) only, and where there is an `exc:ref` element found only in "B" (KG). The paragraph text contents are mostly the same, but "A" (NJ) adds a whitespace before the `exc:ref` and an "A"-only `exc:change-sfs-grp` structure. In this particular case, the merge pipeline adds the attributes as-is, adds the `exc:ref` markup since this is a refer-

[7]And actually a pipelining mechanism, too, which we chose not to use.

ence to KG-only extended editorial notes, and also adds the `exc:change-sfs-grp` NJ-only structure since this is used for print publishing.

I wrote an XProc to run XML Compare on the sources converted to EXC XML. First, the pipeline ran an XSLT to determine which source files that matched each other and which only existed in one source.

Note

How do you determine what source files are the same? Swedish laws are identified by an *SFS number*, a unique identifier for a law, present as a root element attribute in both sources, so that's what the XSLT used.

The result was a list of matching A and B files used as an input to XML Compare, ran using a Delta XML extension step for the XProc engine used, XML Calabash[6][7].

Note

XML Compare can optionally output an HTML representation of the diffed A and B files, which proved helpful when discussing the merge with various project stakeholders.

Note

The other output of that initial XSLT run by the XProc was a list of *unmatched files*, in other words, entire documents present only in one source.

Once diffed, the differences need to be addressed. Again, this was an XProc pipeline running a total of 51 XSLT steps, some of which addressed various aspects of the diff while others up-converted[8] and manipulated diffs.

The merge pipeline addresses a number of situations, among them the following:

- KG-only and NJ-only content that should always be added. Among these are the extended KG editorial notes and any references to those, content intended only for the printed NJ law book, and, of course, any law paragraph versions present in one source but not the other.
- Both sources include short editorial notes and comments. Typically, they explain why and when a certain piece of content was amended, repealed, etc. As both sources have them and they essentially describe the same things, but NJ's are far more extensive than KG's, it was decided that we keep NJ's and discard KG's.

[8]Or rather, sideways conversion; this was a question of fixing problems in the diffed semantics, not creating new semantics altogether. For more, see the issues section.

- Commonly, both sources rely on *auto-linking* — adding links through pattern-matching in the publishing process — for citations[9], but both also add manual citations using cross-reference markup. This resulted in one source using markup for a citation while the other only had a text-based reference. The merge pipeline recognises these differences and defaults to adding the markup, if possible.
- List numbering. If one source uses manual lists (list markup with manual labels) and the other some kind of ordered markup (such as decimal, with the publishing process adding the labels), a step prefers the ordered list types and removes the manual labels.
- Layout hacks in one source. Sometimes, one source would use a “layout hack” to achieve a certain type of formatting. For example, an author might desire the first text paragraph following a list to use the same indentation as the preceding last list item. This might achieved using a manual list type with the last item's label left empty or by adding a second text paragraph inside that last list item.

  The other source, on the other hand, would (correctly) add the text paragraph in question after the list. This would cause a diffing problem, with content added to the list in one source and missing after it in the other.

  A number of the pipeline steps handle permutations of this basic problem, generally adding the offending text paragraph after the list.
- Some differences happen because the sources take a very different approach to marking up the same content.

  For example, one might use a chapter structure to group content in while the other insert running headers between the blocks of content to be grouped. The pipeline doesn't always have the answers — it doesn't know which markup version is correct — so there is a config file listing the preferred versions of content based on specific document IDs. The pipeline looks up the config and deals with the differences accordingly.

Of course, there are quite a few other types of issues; there is a reason for those 51 steps.

3.5. EXC to KG++

In comparison, the EXC to KG++ conversion was a relatively simple task. The pipeline is a mere 15 steps, mostly because the difficult parts had all been addressed already, and because the exchange DTD and the KG++ authoring DTD are closely related — the latter is, with few exceptions, a stricter subset of the exchange DTD.

[9]These are cross-references to law paragraphs, caselaw, and so on.

3.6. KG++ to EXC

Again, the KG++ to EXC pipeline to support the NJ legacy publishing platforms was a simple task, consisting of only 13 steps.

> **Note**
> At the time of this writing, the EXC to old NJ format pipeline is being written, but it's not finished yet.

3.7. Unit Tests

Pipelines, of course, make it easier to limit each step to a well-defined, single task, thus allowing every XSLT stylesheet to be short (well, shortish, at least) and readable. Still, I aimed to write XSpec[9] unit tests for as many transformations as possible. In addition to helping me define a suitable scope for each step, XSpec tests would also quickly pick up my more stupid mistakes.

> **Note**
> If you're reading this while thinking “Duh! Obviously!”, good for you. I know the value of XSpec tests when writing XSLT but still ignore them from time to time, usually out of laziness and hubris. And more often than not, I regret both before long.

I did write an XProc library step that, for each test listed in a manifest file similar to the XSLT manifest, ran the test associated with a specified step. This takes a long time when converting 8,000+ documents and is usually not needed. *What's important is that you start by writing your tests and don't declare the step as done until your tests all pass!*

4. Issues

I encountered numerous issues along the way, but since describing them all here would probably cause a book-length paper, I'll just list some of the more interesting ones.

4.1. eXist-DB to the Rescue!

One of the first things I did when starting this project was to upload both sources to an XML database, *eXist-DB*. I spent time indexing the lot to make the DB as quick as possible, and then, whenever I wondered about the variations in usage of a structure or the frequency of an attribute value, or something else, I'd query the database.

This was almost literally a lifesaver.

I started out with some light queries, but soon, I also used the DB to produce reports and side-by-side comparisons of structures to be reviewed by editors. Without the database, I wouldn't be writing this paper now; I wouldn't be that far along in the project.

4.2. Up-conversion

In principle, before a comparison can take place, the semantics of the sources in EXC format need to be about the same. This, of course, means that most up-conversion needs to be done in the initial conversions to that format. The SFS merge pipeline can handle some differences in semantics but radically different approaches need to be settled before the sources are compared with each other.

4.2.1. Mixed Content

A major headache was the looseness of the block-level KG content models. They allowed mixing textual content with tables, lists, and other typically block-level content (for a simple example, see the KG list markup in Section 3.2). This most likely started out as an oversight in the DTD, but its use soon became widespread.

The equivalent content in the NJ sources had none of this; their block level was neatly separated from inline level.

The up-conversion of the KG block-level confusion was far from trivial. The use was widespread, typically with various container elements mixing text, cross-reference markup, lists, emphasised text, and CALS tables. The container elements themselves could be anything from sections to CALS table entries.

The solution was to add wrappers around the text nodes before separating them from their sibling block-level elements. This was complicated by the presence of inline elements that, of course, should be inside the new text wrappers rather than next to them. But this is the sort of thing `xsl:for-each-group` is for:

```
<xsl:template
    match="textstycke[(table or
        list or
        grafikgrupp or
        tabellingress or
        exc:group[@role='referensgrupp']) and
        not(textblock)] |
        stycke[(table or list or grafikgrupp or
        tabellingress) and not(textblock)]"
    mode="SFS-KG2EXC_BLOCK-LEVEL">

    <xsl:variable name="element-type" select="name(.)"/>
    <xsl:variable name="attrs" select="@*"/>
```

```
    <xsl:for-each-group
        select="node()"
        group-starting-with="table | list | tabellingress | grafikgrupp |
            exc:group[@role='referensgrupp'] |
            notreferens[not(preceding-sibling::node()[1][self::text()])]
|
            eksternref[not(preceding-sibling::node()[1][self::text()])]
|
            referens[not(preceding-sibling::node()[1][self::text()])] |
            footnote[not(preceding-sibling::node()[1][self::text()])] |
            format | sup[not(preceding-sibling::node()[1]
[self::text()])] |
            text()[preceding-sibling::*[1][self::table or self::list or
self::exc:group]]">

        <xsl:choose>

            <xsl:when
                test="self::format[not(table) and not(list)] or
                    self::text() or
                    self::sup or
                    self::notreferens or
                    self::eksternref or
                    self::referens or
                    self::footnote">
                <xsl:element name="{$element-type}">
                    <xsl:copy-of select="$attrs"/>
                    <xsl:apply-templates select="current-group()"
mode="SFS-KG2EXC_BLOCK-LEVEL"/>
                </xsl:element>
            </xsl:when>
            <xsl:otherwise>
                <xsl:apply-templates select="current-group()" mode="SFS-
KG2EXC_BLOCK-LEVEL"/>
            </xsl:otherwise>
        </xsl:choose>

    </xsl:for-each-group>

</xsl:template>
```

I'm not going to walk through the above; this is merely to show the kind of solution I'd typically employ to sort out the confusing mix of block-level and inline elements and text nodes. Several similar steps walked through similar problems in different contexts.

4.2.2. Running Headers

Another major issue, this time with both sources, was that rather than using an explicit structure to indicate section and subsection levels, they'd simply insert a *running header* with formatting instructions in attributes to indicate the heading's size:

```
<header type="3">Running Header 1</header>
<para>Section content</para>

<header type="4">Running Header 2</header>
<para>Subsection content</para>

<header type="4">Running Header 3</header>
<para>More subsection content</para>

<header type="3">Running Header 4</header>
<para>New section content</para>
```

There is no actual section or subsection structure, only running headers with `type` attributes implying a section and subsection structure. For the author, the values indicate relative font sizes; “3”, here, implies a larger size than “4” and thus, a section rather than a subsection.

When formatted, the reader will interpret these headers as headings and subheadings that group the document contents. This is how the human brain works; we pattern-match and we interpret implied structures as explicit.

To complicate things, both source DTDs would also allow explicit part and chapter structures, and allow them *mixed* with the running headers:

```
<!ELEMENT body  (part | chapter | header | paragraph)* >
```

The above is a translated approximation of the model; the source DTDs are in Swedish and the models are more complex than this. Note that both `part` and `chapter` have similar models, allowing running headers mixed with both block- and section-level content.

If both companies had been consistent with their use of the running headers (and any explicit grouping elements), we'd not have much of a problem. We'd simply map the use of the running headers and merge them. In reality, however, one source might use a part and chapter structure while the other chose to use running headers to imply the same.

Here, the solution was a mix of techniques. Apart from a few documents that were simply too different to be merged by the pipeline[10], the solution I came up with was a mix of up-conversion to recursively group running headers followed by block-level elements and, possibly, “lower-level” running headers and more

[10]Forcing me to write a step that cherry-picked structures based on the document's SFS number.

block-level content, add various to the groups when treated as such, and then down-convert right before the diff and the merge.

So, for example, an up-conversion might result in something like this:

```
<group>
    <header level="3">...</header>
    ...
    <group>
        <header level="4"></header>
        ...
    </group>
    ...
</group>
```

This would allow me to manipulate the implied section hierarchy as such, but then flatten the thing again so it could more easily be compared to the equivalent document in the other source. The flattened XML might then look like this:

```
<group-divider level="3"/>
<header level="3">...</header>
...
<group-divider level="4"/>
<header level="4"></header>
...
...
```

The `group-divider` element would contain any information I'd need to recreate the recursive group later, after comparing and merging two flattened structures.

Again, if the sources had the same number of running headers, this would be much easier. Unfortunately, one source might have *additional* running headers, which would then have to be added to the merged XML, adding to the total number of headers and subheaders[11] in the result.

As I strongly believe that actual, explicit, section structures are much preferable, a later step, after the merge, recreates the recursive grouping for future authoring[12].

4.2.3. Manual Lists

Another required up-conversion is the differing list types problem. For example, NJ might have use an ordered list type like this[13], the processing assumption being that the list item labels are generated when publishing:

```
<list type="decimal">
    <item>Apples</item>
```

[11]And possibly adding extra *structural* complications such as extra levels or broken levels.
[12]The new DTD doe not allow running headers anywhere, which has been, um, a matter of debate.
[13]Obviously, I'm faking both tag names and the content here. Duh.

```
    <item>Oranges</item>
    <item>Bananas</item>
</list>
```

KG's take on the same list might be a manual list, with the list item labels part of the content:

```
<list type="manual">
    <item>1. Apples</item>
    <item>2. Oranges</item>
    <item>3. Bananas</item>
</list>
```

Of course, sometimes the manual list is there for an actual reason, such as:

```
<list type="manual">
    <item>1. Apples</item>
    <item>1A. Nuts</item>
    <item>2. Oranges</item>
    <item>3. Bananas</item>
</list>
```

The “1A.” label cannot be automatically generated, of course. The usual reason, however, is simply that the author prefers a label with a different appearance.

When converting to the exchange content, I have steps in place that detect manual lists that really shouldn't have been manual in the first place, and can do the up-conversion with relative ease. But authors are sometimes more creative than that.

Let's say that the list needs to be followed by an explanatory sentence. The NJ source looks like this:

```
list type="decimal">
    <item>Apples</item>
    <item>Oranges</item>
    <item>Bananas</item>
</list>
<p>This is my favourite food.</p>
```

The KG equivalent, however, is this:

```
<list type="manual">
    <item>1. Apples</item>
    <item>1A. Nuts</item>
    <item>2. Oranges</item>
    <item>3. Bananas</item>
    <item>This is my favourite food.<item>
</list>
```

A quick glance suggests that this is a list with four items. What we have here, however, is a manual list type used for formatting purposes: that last item will have the same margins as the actual list items, which is what the author wants.

This is a lot more difficult to get right. The initial conversion to EXC takes care of the manual numbering, but leaves behind the wrong number of list items. Those I can only handle once I've compared the sources and added the Delta XML markup, and even then it takes a couple of steps as there are variations on this basic theme[14].

For more variation of this theme, see Section 4.3.1.

4.3. Tag Abuse

Both sources had widespread tag abuse. For example, what was marked up as a list in one source might be marked up as ordinary text paragraphs in the other, and sometimes, ordinary text paragraphs were used in place of subheadings. What sets tag abuse apart from the up-conversion requirements as discussed above is mostly a matter of definition...

4.3.1. Fake Lists

A variation of the manual list up-conversion was the numerous *fake lists* also present. So, instead of list markup, one source would simply have something like:

```
<p>1. Apples</p>
<p>2. Oranges</p>
<p>3. Bananas</p>
```

Now, while this is only a problem if one source uses actual list markup and the other doesn't, I nevertheless decided to take the fix-always approach after some querying of the sources in eXist-DB. See, unlike (some? many? most?) other types of content, legal content is very formalistic. It is a rare thing to use a number to start a sentence with, even rarer to always add a full stop (or some other delimiter) after it, and almost unheard of to have two or more text paragraphs repeat the same pattern.

I was right. At least, the diff and the subsequent merge suggest I was. The problem, instead, was normally a variation of the stray paragraph at the end of the list, similar to what I encountered with the manual lists (see Section 4.2.3). The fix, then, was more or less the same: first, up-convert to a list, then fix the discrepancies in the merge pipeline.

[14]For example, there might be two text paragraphs rather than a single one, which immediately makes the fix more difficult.

4.3.2. Fake Headings

Not entirely uncommon was to use an ordinary emphasised paragraph as a running header[15], adding emphasis to highlight the "header nature":

```
<p><emph>Header</emph></p>
```

This was a minor issue, and easy to fix.

4.4. Preferred Content

In some cases, we'd not merge at all, but instead prefer one source over another, depending on various requirements. For example:

- Early on, it was decided that the NJ editorial comments would be used rather than the KG equivalents, whenever given a choice. Of course, anything that was KG only (versions missing in the NJ equivalent or entire documents) still needed the KG comments.
- KG has a type of extended commentary that was always to be included. This was relatively uncomplicated since the commentary itself was out of line and referenced by `EMPTY` elements inline. I'd add the references inline in an early merge step, regardless of other considerations, and then moved on to merging the content surrounding the references.
- Amendment information is present in every law and explains when a piece of legislation comes into force, by using what law, and what, if anything, it replaces. Here, the decision was again to use the NJ amendment information.

Of course, the general rule was to always add a missing law paragraph version, just as we'd always add a document only present in one source.

4.5. Manual Selection

Sometimes, there'd be deviations to the norm, content that was so different between the sources that there was no way to do an automated merge. For example:

- Many documents have supplements, extra information that is usually not provided directly by the government office. These can be anything from forms to pieces of EU legislation. As such, they were often quite different between the sources and it was out of the question to attempt an automated merge. Instead, the editorial teams spent time cherry-picking the supplements based on a side-by-side comparison I'd generated for the purpose in eXist-DB.

[15]Why not a running header tag? I'd suspect that the formatting requirements were different, or that there was no "formal" equivalent to the header in the sources. This was often the case with supplements, additions to the laws outside what the government provides, as well as various editorial comments.

- Some documents, especially older ones, might be tagged so differently that an automated merge wouldn't be possible. Usually, this resulted from tag abuse, but sometimes also because of poor original scans.

  In these cases, I ended up adding early conversion steps that would apply custom conversion to the documents, based on their SFS numbers rather than some general rule.

4.6. Diffing and Merging Problems

4.6.1. Identifying Normative Content

When you're comparing sources that supposedly have the same base, along with some definable differences, it helps a lot if you can tell the compare process that certain nodes are intended to be the same. For XML Compare, you tell the application by adding `deltaxml:key` attribute values to any nodes that are the same in both sources.

In SFS documents, some semantics are *normative* and can therefore safely be assumed to be the same in both sources. For example, law paragraphs are normative, as are various subparagraph constructs below them, and chapters above them. Laws change, however, and law paragraphs are frequently amended or repealed, and so saying "Chapter 2, §3 in SFS 1999:913 in KG is equivalent with Chapter 2, §3 in SFS 1999:913 in NJ" is not enough.

Laws are amended by other laws. A complete law will contain any number of historical versions, one current version, and possibly one future version, of any versioned component that is subject to change, like so:

```
<paragraph
    logid="SFS1962-0700.K2.P3">
    <desig>3 §</desig>

    <paragraph-version
        valid-from="20190208"
        valid-to="40000101"
        change-sfs="SFS2019-0021"
        status="future"
        logid="SFS1962-0700.K2.P3-20190208-SFS2019-0021">
        ...
    </paragraph-version>

    <paragraph-version
        valid-from="19990913"
        valid-to="20190208"
        change-sfs="SFS1999-0851"
        status="current"
        logid="SFS1962-0700.K2.P3">
```

```
        ...
    </paragraph-version>

    <paragraph-version
        valid-from="19650101"
        valid-to="19990913"
        change-sfs="SFS1962-0700"
        status="historical"
        logid="SFS1962-0700.K2.P3-19650101-SFS1962-0700">
        ...
    </paragraph-version>
</paragraph>
```

This is *3 §* of some law, shown here with all its available versions. Each have a valid-from and a valid-to date, indicating when the version came into force and when it was amended. The instrument used to amend it is identified by the `change-sfs` attribute. Here, the oldest version came into force on 1 January 1965 and the instrument used was the actual law, SFS 1962:700. The paragraph was then amended by SFS1999:851, and that amendment came into force on 13 September 1999. The `status` identifies it as currently being in force, but there is also a future version of the paragraph, amended by SFS 2019:21, that will come into force on 8 February 2019.

The nodes all also have a `logid` attribute. All but the currently in force version of the paragraph have `logid` values consisting of a base identifier ("SFS1962-0700.K2.P3") that identifies the current SFS document ("SFS1962-0700"), the current chapter ("K2"), and the current paragraph ("P3"), followed by the valid-from date (for example, "19650101") and the amending SFS number (for example, "SFS1962-0700"). The `logid` is not an actual ID — the values are not unique, even here — but as the paragraphs and their versions are normative, their `logid` attributes can be used as keys.

Note

The `logid` values used here are a KG convention, not anything required by the content as such. The NJ equivalent was far less formalistic, so I ended up generating `logid`-like identifiers for both content sets when converting them to EXC format. This was the only way to be sure that the same "IDs" were used by both sources.

The last step in the pipelines converting the sources to the exchange format adds `deltaxml:key` attribute values to all normative elements. In theory, this allows us to uniquely identify content that are the same in both sources.

In theory.

4.6.2. Versioning Problems

In a perfect world, if both companies had saved each and every version of the Swedish Code of Statutes since they were first written[16] by lawmakers, being able to identify a paragraph number, when it came into force, and the amending law is enough for unique identification. Unfortunately, we do not live in a perfect world:

- Not every version was saved. This becomes obvious when studying the older legislation. There are huge gaps in anything older than, say, 1990.
- When transcribing and marking up the PDF sources from the government office, sometimes the valid-from date is not yet known, so an issued-date or a published-date is used instead.
- Frequently, historical versions of paragraphs in older, but important, laws were added later, sometimes decades later[17]. As the valid-from dates of that law varied — not everything came into force at the same time — sometimes these dates were unknown and so a later date was used instead.
- For quite a few laws in the Karnov sources, the valid-from date inserted, when the historical version's actual in-force date was unknown, was 1066-10-14. This is known as the "Battle of Hastings problem" in the SFS merge project.

  Norstedts, thankfully, ran a large project some years ago to correct any dates that were known to be incorrect. When merging SFS documents that were present in both sources, I was able to use the NJ dates rather than the Battle of Hastings date. Unfortunately, quite a few versions are unique to Karnov and so still have these dates.
- And here's an interesting date problem: none of the KG SFS source document has valid-to dates, only valid-from. Presumably, this started a long time ago as an assumption in the SQL database where the documents live. A version is valid until the next one becomes valid, so there's no need to add the valid-to date.

  This assumes, of course, that every version is available, which is simply not the case. If a version is missing, various diffing problems can ensue if NJ's corresponding document has a different set of versions.

There are many variations of this basic theme — a missing, or wrong, valid-from date. As the Delta XML key values depend on the date, different valid-from dates results in the merge producing two different versions of what should have been a single version.

You might now be wondering why the amending SFS number (the `change-sfs` attribute in the example in Section 4.6.1) isn't enough. The problem is that an

[16]The oldest SFS document I know of is from 1540.

[17]For example, *Criminal Law*, that first came into force in 1962, has been edited in late 1980s and early 1990s to include historical versions, when publishing on CD-ROMs first became a thing.

amendment might amend paragraphs on different dates, in phases. Simply put, § 5 might be amended first and § 6 six months later. Without dates, both amendments would come into force at the same time. Therefore, the date is required.

4.6.3. Versioning Problems, Pt II

While both sources versioned law paragraphs and some other content on the same level, NJ also used a similar approach to version chapters. A new chapter version, basically, happened by definition — an amendment would simply amend an entire chapter, copying everything from the old chapter to the new while changing the statuses of all versioned content of the old chapter to "historical". This might happen for a reason as simple and straight-forward as the chapter getting a new title.

KG, however, did not version chapters, choosing instead to version their titles, as well as titles in a number of other contexts. An amendment SFS number and an accompanying in-force date changing the chapter version at NJ would thus be equivalent to an amendment version and in-force date changing a chapter title's version at KG[18].

Merging the two produced strange results, however, and eventually I decided to use XQuery and the eXist-DB sources to produce a report of every single occurrence of a chapter version in NJ SFS documents, including, of course, change SFS numbers and in-force dates but also the chapter version contents including *their* versions, change SFS numbers, and in-force dates, and then use the report to generate chapter versions to the KG content in EXC format. Once that was done, merging chapter-level contents went without further incident.

4.6.4. Other Diffing Problems: The Merge As A Quality Assurance

Sometimes, I'd end up with a merged document that was invalid because the sources had marked up what was supposedly the same content very differently:

- One source used a CALS table to simulate a list[19] while the other used proper list markup. There's usually no way to merge this sort of thing automatically, at least not without considerable effort, so the solution here was to simply pick the preferred source and move on.
- Often, I'd find about tag abuse when running the merge. For example, if one source used running headers to simulate, say, a chapter structure while the other used actual chapter markup, the merge would be where I first discovered the problem. Usually I'd query my eXist-DB sources to find out how

[18]As chapters are considered to be normative, I'd say that NJ's approach is closer to the intended semantics.

[19]Presumably, you get beautiful margins...

common the problem was and then fix it in the initial source-to-exchange conversion pipelines. If the problem

5. Conclusions

The project is still ongoing as I write this, but I can offer a number of conclusions here:

- It's doable! To a big part, this is because we can base a lot of the merge decisions on the fact that we actually have (or at least should have) the same main source content. Once the respective sources themselves have been sufficiently tweaked, it is possible to diff and merge everything while neatly separating any additional material from that main content.
- With any large data set such as SFS, there are bound to be cases requiring manual intervention. In our case, out of the thousands of documents, we had perhaps two or three dozen documents that we had to add to an exception list when merging; that exception list, basically an XML config file for the conversion, simply declared preferred handling for the listed documents as identified by their SFS numbers.
- As with mostly every other technical implementation, many issues are cultural rather than technical. The new authoring DTD, called KG++ throughout this paper, has already generated more work than any other single piece of work in the project, mostly for reasons unrelated to any technical requirement.

5.1. Why Not XQuery?

“Why not do it in XQuery and an XML database? You already use eXist-DB, don't you? I'm sure at least some of your fixes to the sources are better done in eXist, right?”

I wanted an easily repeatable process and something that would always produce debug information in the same way for each and every fix. While it's certainly possible to run XQuerys in XProc, accessing eXist-DB from an XProc that needs to run on a file system is currently an extra level of pain[20].

Short Glossary

Just a short list of terms and abbreviations. I don't know about you, but I always hate it when I can't immediately figure out the meaning of an abbreviation. Hence this list.

[20]And running everything in eXist-DB is currently not an option, not if you want to use XProc.

Exchange DTD
: The exchange DTD format used to describe the “sum” of the respective source DTD semantics. Quite a useful thing, really.

Karnov Group
: Karnov Group, Danish legal publisher.

KG++
: Nickname for the new authoring DTD for the merged SFS documents.

Norstedts juridik
: Norstedts juridik, a Swedish legal publisher. Bought by KG.

Swedish Code of Statutes
: The official law code of Sweden, comprising thousands upon thousands of distinct statutes.

Bibliography

[1] *Swedish Code of Statutes* https://en.wikipedia.org/wiki/Swedish_Code_of_Statutes

[2] *Svensk författningssamling* https://www.svenskforfattningssamling.se/english.html

[3] *Difference And Compare XML Files With XML Compare* https://www.deltaxml.com/products/compare/xml-compare/

[4] *XProc: An XML Pipeline Language* https://www.w3.org/TR/xproc/

[5] *XProc Tools* https://github.com/Corbas/xproc-tools

[6] *XML Calabash* http://xmlcalabash.com/

[7] *XML Calabash Delta XML extension step* https://github.com/ndw/xmlcalabash1-deltaxml

[8] *Up and Sideways: RTF to XML* https://doi.org/10.4242/BalisageVol20.Nordstrom01

[9] *XSpec* https://github.com/xspec/xspec

JLIFF, Creating a JSON Serialization of OASIS XLIFF

Lossless exchange between asynchronous XML based and real time JSON based pipelines

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

Phil Ritchie
Vistatec
<phil.ritchie@vistatec.com>

Robert van Engelen
Genivia
<engelen@genivia.com>

Abstract

JLIFF [10] is the JSON serialization of XLIFF. Currently [10] only exists as a reasonably stable JSON schema [11] that is very close to being a full bidirectional mapping to both XLIFF 2 Versions. XLIFF is the XML Localization Interchange File Format. The current OASIS Standard version is XLIFF Version 2.1 [21]. The major new features added to [21] compared to XLIFF Version 2. 0 [20] are the native W3C ITS 2.0 [8] support and the Advanced Validation feature via NVDL and Schematron. This paper describes how XLIFF was ported to JSON via an abstract object model [14]. Challenges and design principles of transforming a multi-namespace business vocabulary into JSON while preserving lossless machine to machine interchange between the serializations are described in this paper. While we do explain about the Internationalization (I18n) and Localization (L10n) business specifics, we are also striving to provide general takeaways useful when porting XML based vocabularies into semantically and behaviorally interoperable JSON serializations.

Keywords: inline data model, UML, JLIFF, JSON, JSON-LD, W3C ITS, XLIFF, Internationalization, I18n, Localization, L10n, metadata, multi-namespace, namespaces, mapping, roundtrip, lifecycle, multilingual content

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

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

1. Introduction

In this paper and XML Prague presentation, we will explain about *JLIFF* [10], the JSON serialization of *XLIFF*. JLIFF is designed to start with the 2.0 version number and bidirectional mapping for both XLIFF 2.0 [20] and XLIFF 2.1 [21] is being built in parallel in the initial version. The design is extensible to support any future XLIFF 2.n+1 Version. The *XLIFF 2* series has been designed to be backwards and forwards compatible. XLIFFF 2.n+1 versions can add orthogonal features such as the Advanced Validation added in [21] or the Rendering Requirements [25] to be added in XLIFF 2.2. All XLIFF 2.n specifications share the core namespace `urn:oasis:names:tc:xliff:document:2.0`.

OASIS XLIFF OMOS TC only recently started developing the [10] prose specification that should be released for the 1st public review by April 2019. It is however clear that the specification largely mimics the logical structure of the XLIFF 2 specifications. JLIFF is designed to mirror XLIFF including its numerous modules, albeit via the abstract object model. The modular design of XLIFF 2 makes critical use of XML's namespaces support. Each XLIFF 2 module has elements or attributes defined in namespaces other than the core XLIFF 2 namespace. This allows the involved agents (conformance application targets) to handle all and only those areas of XLIFF data that are relevant for their area of expertize (for instance *Translation Memory* matching, *Terminology*, *Text Analysis* or entity recognition, *Size and Length Restrictions*, and so on). Now, how do you handle multi-namespace in JSON that doesn't support namespaces? This is covered in The design of JLIFF.

The data formats we are describing in this paper are for managing *Internationalization* and *Localization* payloads and metadata throughout the multilingual content lifecycle. Even though corporations and governments routinely need to present the same, equivalent, or comparable content in various languages, *multilingual content* is usually not consumed in more than one language at the same time by the same end user. Typically the target audience consumes the content in their preferred language and if everything works well they don't even need to be aware that the monolingual content they consume is part of a multilingual content repository or a result of a *Translation*, *Localization*, or cultural adaptation process.

Thus *Multilingualism* is transparent to the end user if implemented properly. To achieve end user transparency the corporations, governments, inter- or extra-

national agencies need to develop and employ *Internationalization*, *Localization*, and *Translation* capabilities. While *Internationalization* is primarily done on a monolingual content or product, *Localization*, and *Translation* when done at a certain level of maturity -- as a repeatable process possibly aspiring to efficiencies of scale and automation -- requires a persistent *Bitext* format. Bitext in turn requires that *Localizable* or *Translatable* parts of the source or native format are *Extracted* into the Bitext format, which has provisions for storing the Translated or Localized target parts in an aligned way that allows for efficient and automated processing of content during the Localization roundtrip.

Our paper presented to XML Prague 2017 [3] made a detailed introduction of XLIFF ([22] as the then current predecessor of [21] backwards compatible with [20]) as the open standard *Bitext* format used in the Localization industry. This paper describes how the complete open transparent Bitext capability of XLIFF can be ported to JSON environments using the JLIFF format. We also demonstrate that JLIFF and XLIFF can be used interchangeably, effectively allowing to switch between XML and JSON pipelines at will.

2. Lay of the land

2.1. I18n and L10n Standards

The foundational Internationalization Standard is of course [18] along with some related Unicode Annexes (such as [17]). However, in this paper we are taking the Unicode support for granted and will be looking at the domain standards W3C ITS 2.0 [8] and OASIS XLIFF [21] along with its upcoming JSON serialization [10] that are the open standards relevant for covering the industry process areas outlined in the second part of the Introduction.

For a long time, XML has been another unchallenged foundation of the multilingual content interoperability and hence practically all Localization and Internationalization standards started as or became at some point XML vocabularies. Paramount industry wisdom is stored in data models that had been developed over decades as XML vocabularies at OASIS, W3C, LISA (RIP) and elsewhere. Although ITS is based on *abstract metadata categories*, *W3C ITS 1.0* [7] had only provided a specific implementable recommendation for XML. The simple yet ingenious idea of ITS is to provide a reusable namespace that can be injected into existing formats. Although the notion of a namespace is not confined to XML, again [7] was only specifically injectable into XML vocabularies.

[8] provides local and global methods for metadata storage not only in XML but also in [6], it also looked at mapping into non-XML formats such as [15], albeit in a non-normative way. Because native HTML does not support the notion of namespaces, [8] has to use attributes that are prefixed with the string `its-` for

the purpose of being recognized as an HTML 5 module. In [10], we are using `its_` to indicate the ITS Module data.

[8] also introduced many new metadata categories compared with [7]. ITS 1.0 only looked at metadata in source content that would somehow help inform the Internationalization and Localization processes down the line. ITS 2.0 brought brand new and sometimes complex metadata categories that contain information produced during the localization processes or during the language service transformations that are necessary to produce target content and are typically facilitated by Bitext. This naturally led to a non-normative mapping of [8] to [19] and to [20] (that was then in public reviews). Thus ITS 2.0 became a very useful extension to XLIFF. And here comes the modular design that allows to turn useful extensions into modules as part of a dot-release. Not only module data is better protected but also describing a data model addition as part of the broader spec gives an opportunity to tie lots of loose ends that are at play when using only a partially formalized mapping as an extension.

One of the main reasons why [20] is not backwards compatible with [19] is that the OASIS XLIFF TC and the wider stakeholder community wanted to create XLIFF 2 with a modularized data model. [20] has a small non-negotiable core but at the same time it brings 8 namespace based modules for advanced functionality. The modular and extensible design aims at easy production of "dot" revisions or releases of the standard. *XLIFF Version 2.0* [20] was intended as the first in the future family of backwards compatible XLIFF 2 standards that will share the maximally interoperable core (as well as successful modules surviving from 2.0). XLIFF 2 makes a distinction between modules and extensions. While module features are optional, *Conformant XLIFF Agents* are bound by an absolute prohibition to delete module based metadata (MUST NOT from [4]), whereas deletion of extension based data is discouraged but not prohibited (the SHOULD NOT normative keyword is used, see [4]). The *ITS Module* is the biggest feature that was requested by the industry community and approved by the TC for specification as part of [21].

So in a nutshell, the difference between XLIFF 2.1 and XLIFF 2.0 can be explained and demonstrated as the two overlapping listings of namespaces.

Example 1. Namespaces that appear both in XLIFF 2.1 and XLIFF 2.0

```
urn:oasis:names:tc:xliff:document:2.0          <!-- Core1 -->

urn:oasis:names:tc:xliff:matches:2.0           <!-- Translation
Candidates Module2 -->

urn:oasis:names:tc:xliff:glossary:2.0          <!-- Glossary Module3 -->
```

[1] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#core

[2] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#candidates

```
urn:oasis:names:tc:xliff:fs:2.0                <!-- Format Style
Module4 -->

urn:oasis:names:tc:xliff:metadata:2.0          <!-- Metadata Module5 -->

urn:oasis:names:tc:xliff:resourcedata:2.0      <!-- Resource Data
Module6 -->

urn:oasis:names:tc:xliff:sizerestriction:2.0   <!-- Size and Length
Restriction Module7 -->

urn:oasis:names:tc:xliff:validation:2.0        <!-- Validation Module8
-->
```

Example 2. Namespaces that appear only in XLIFF 2.1

```
http://www.w3.org/2005/11/its                  <!-- ITS Module9 -->

urn:oasis:names:tc:xliff:itsm:2.1              <!-- ITS Module10 -->
```

Example 3. Namespaces that appear only in XLIFF 2.0

```
urn:oasis:names:tc:xliff:changetracking:2.0    <!-- Change Tracking
Module11 -->
```

Apart from the 11 listed namespaces, both *XLIFF Core* and the *W3C ITS namespaces* reuse the `xml` namespace. This is still not all namespaces that you can encounter in an *XLIFF Document*. XLIFF 2 Core defines 4 element extension points (`<file>`, `<skeleton>`, `<group>`, and `<unit>`) and 4 more attribute extension points (`<xliff>`, `<note>`, `<mrk>`, and `<sm>`). Most of XLIFF's modules are also extensible by elements or by attributes. We will explain in the JLIFF design section how we dealt with the inherent multi-namespace character of XLIFF. Both module and extension data are allowed on the extension points with some notable distinctions and exceptions. Module attributes can be added not only at the above listed 4 extension points but can be also specifically allowed on `<pc>`, `<sc/>`, and `<ec/>`. Generally module namespaces based data is only considered Module (and hence

[3] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#glossary-module
[4] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#fs-mod
[5] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#metadata_module
[6] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#resourceData_module
[7] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#size_restriction_module
[8] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#validation_module
[9] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#ITS-module
[10] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#ITS-module
[11] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#changeTracking_module

absolutely protected) data when it appears on the extension points where it is explicitly listed in the prose specification which corresponds to where it is allowed by core NVDL and Schematrons. Core xsd is not capable of making this distinction.

2.2. The Notion of Extracting XLIFF Payload from Native Formats

Best practice for *Extracting* native content into [21] has been recently specified as a deliverable of the GALA TAPICC Project in [23], see this publicly available specification for details on Extracting XLIFF. We will provide a condensed explanation of the *Extraction* concept here. The most suitable metadata category to explain the idea of Extraction from the native format with the help of ITS is *Translate*. This is simply a Boolean flag that can be used to indicate Translatability or not in source content.

Example 4. Translate expressed locally in HTML

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

Example 5. Translate expressed locally in XML

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

Since it is not always practically possible to create local annotations, or the given source format or XML vocabulary has elements or attributes with clear semantics with regards to some Internationalization data categories such as Translate, in most cases, ITS 2.0 also defines a way to express a given data category globally.

Example 6. Translate expressed globally in XML

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

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

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

Example 7. XLIFF Core @translate on a structural leaf element

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

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

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

The same snippet could be also represented like this:

Example 8. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
        &lt;segment&gt;
          <source>Translatable text</source>
        &lt;/segment&gt;
 </unit>
```

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

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

Example 9. XLIFF Core @translate used inline

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

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

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

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

Example 10. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
     &lt;segment&gt;
         <source>Text <ph id='1'/></source>
     &lt;/segment&gt;
 </unit>
```

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

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

Example 11. ITS global rule to detect translatability in XLIFF

```
<its:rules version="2.0" queryLanguage="xpath">
       <!-- Rules for Translate -->
       <its:translateRule selector="//xlf:*[@translate='no']"
translate='no'/>
       <its:translateRule selector="//xlf:*[@translate='yes']"
translate='yes'/>
  </its:rules>
```

The above rule will correctly identify all XLIFF nodes that are using the `xlf:translate` attribute with one important caveat, Translatability annotations on pseudo-spans will be interpreted as empty <sm/> nodes. And pseudo-span

Translatability overlaps with Translatability well-formed markup will not be properly interpreted, see [21] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#translateAnnotation.

3. The abstract Localization Interchange Object Model (LIOM)

3.1. The Core Structure

Figure 1 is a UML class diagram rendering of the abstract object model behind XLIFF 2 Core and hence also JLIFF core.

The above can be described in a natural language as follows:

A *LIOM* instance contains at least one file. Each file can contain an optional recursive group structure of an arbitrary depth. Because grouping is fully optional, files can contain only a flat series of units. But also any file or group can contain either a flat structure of units or an array of groups and units (subgroups). Each unit contains at least one sub-unit of the type segment (rather than ignorable). Each sub-unit contains exactly one source and at most one target. Bitext is designed to represent the localizable source structure and only later in the process is expected to be Enriched with aligned target content. The content data type can contain character data mixed with inline elements. It is worth noting that XLIFF even in its XML serialization only preserves a treelike document object model (DOM) down to unit. Inline markup present in the source and target content data can form spans that can and often have to overlap the tree structure given by the well-formed `<mrk>` and `<pc>`, but also notably the structural `<source>`, `<target>`, `<segment>`, and `<ignorable>` tags. The `<unit>` tag separates the upper clean XML well-formed structure from the transient structure of segments where "non-well-formed" pseudo-spans formed by related empty tags (`<sc id="1"/ >` and `<ec startRef="1"/ >`, as well as `<sm id="2"/ >` and `<em startRef="2"/>` pairs) need to be recognized and processed by *XLIFF Agents*. See [21] Spanning Code Usage[12] ("*Agents* MUST be able to handle any of the above two types of inline code representation") or Pseudo-span Warning[13]. Since the equivalence of well-formed versions of the spanning codes `<pc>` and markers `<mrk>` with the above pseudo-spans is defined and described in XLIFF 2 itself, there is no need to include the well-formed versions of the inline tags in the abstract LIOM and non-XML serializations including JLIFF are free to use a fully linear inline data model.

The above class diagram shows that any LIOM instance has four options of logical roots. The original XML serialization, i.e. XLIFF 2, can only use the top level root object according to its own grammar. On the other hand, the abstract

[12] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#spanningcodeusage
[13] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#pseudo-spanWarning

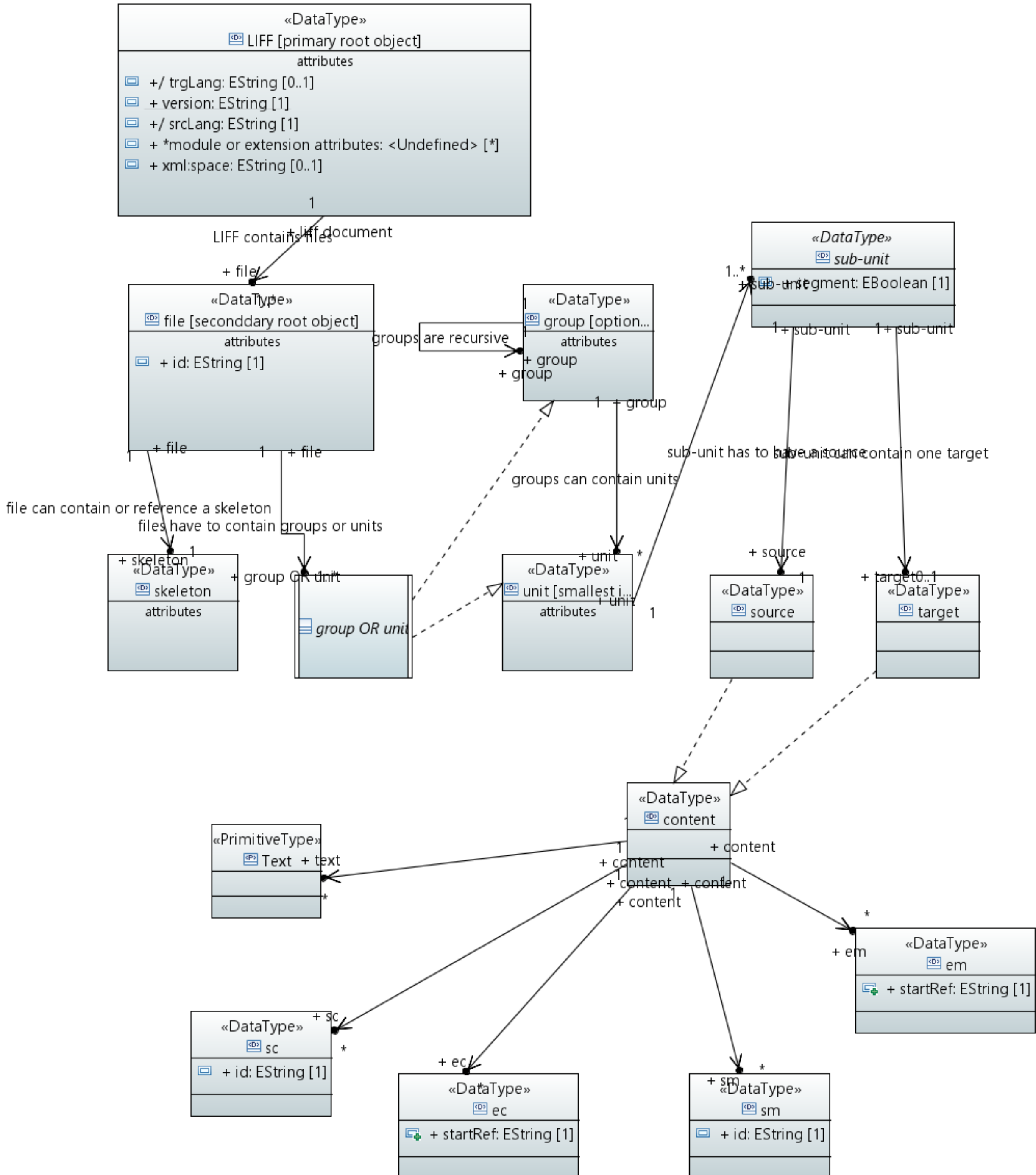


Figure 1. The abstract Localization Interchange Object Model - [14]

object model caters for use cases where LIOM fragments could be exchanged. Such scenarios include real time unit exchange between translation tools such as between a *Translation Management System (TMS)* and a translation or review workbench (browser based or standalone), a TMS and a *Machine Translation (MT)* engine, two different TMSes, and so on.

Based on the above, a LIOM instance can represent a number of *source files*, a single source file, a structural subgroup (at any level of recursion) or a smallest self contained logical unit that are intended for Localization.

The top level wrapper in the XML serialization is the `<xliff>` element, the top level object in the JSON serialization is an anonymous top level object with the required `jliff` property.

Example 12. XLIFF top level element

```
<xliff xmlns="urn:oasis:names:tc:xliff:document:2.0"
xmlns:uext1="http://example.com/userextension/1.0"
xmlns:uext2="http://example.com/userextension/2.0"
version="2.1" srcLang="en" trgLang="fr">
  <file … >
    <group … >
      /arbitrary group depth including 0/
      <unit … >
      [ … /truncated payload structure / … ]
      </unit>
    </group>
  <file>
</xliff>
```

Example 13. JLIFF anonymous top level object

```
{
    "jliff": "2.1",
    "@context": {
      "uext1": "http://example.com/userextension/1.0",
      "uext2": "http://example.com/userextension/2.0"
    },
    "srcLang": "en",
    "trgLang": "fr",
    "files | subfiles | sugbroups | subunits": [ … /truncated payload
structure / … ]
}
```

Comparing the two examples above, it is clear that XLIFF in its original XML serialization doesn't have another legal option but to represent the whole project structure of source files. JLIFF has been conceived from the beginning as the JSON Localization *Fragment* Format, so that top level JLIFF object (`jliff`) can wrap an array of files (within the `files` object), an array of groups or units (within the `subfiles` or the `subgroups` object), or an array of sub-units (within the `subunits` object). Since the data model of a subfile and a subgroup is identical, `subfiles` and `subgroups` are instances of a common JSON schema type

named `subitems`. The `subitem` type object simply holds an array of anonymous group or unit objects.

The `jliff` property values are restricted at the moment to `2.1` or `2.0`. The context property is optional as it is only required to specify [12] context for extensions if present. This is a workaround to declaring the same extensions' namespaces as in the XLIFF top level example. The `srcLang` property is required while the `trgLang` property is only required when `target` objects are present.

3.2. LIOM Modules

3.2.1. LIOM modules originating in XLIFF 2.0

[20] defined 8 namespace based modules, from which 7 survived to [21], see the namespaces listing above. We won't be dealing with the deprecated Change Tracking Module.

The simplest [21] Module is the Format Style Module[14] that consists just of two attributes `fs` and `subFs`. Obviously this very easy to express both in the abstract LIOM and in the JLIFF serialization. The `subFs` property is only allowed on objects where `fs` has been specified. At the same time `fs` values are restricted to a subset (see [21] http:// docs.oasis-open.org/ xliff/ xliff-core/ v2.1/ os/ xliff-core-v2.1-os.html# d0e13131) of [6] tag names. While the Format Style Module only provides 2 properties, it is rather far reaching as these are allowed on most structural and inline elements of XLIFF Core. Information provided through `fs` and `subFs` is intended to allow for simple transformations to create previews. Previews are extremely important to provide context for human translators.

Apart from Format Style, all other Modules define their own elements. In general each Module's top wrapper is explicitly allowed on certain static structure elements and has a reference mechanism to point to a core content portion that it applies to. Glossary Module data can be also pointed to *vice versa* from the Core Term Annotation[15].

The [21] Translation Candidates Module[16] is only allowed on the unit level and serves for storing of locally relevant Translation suggestions. Those typically come from a TM or MT service. The Translation Candidate module reuses the core source and target data model, as the data is designed to be compared with the core contained source content and populate the core target content parts from the module target containers. While this primarily targets human translators selecting suitable translation candidates. Most TMSes have a function to populate or pre-populate core target containers with the module based suggestions based on some decision making algorithms driven by the match metada caried within

[14] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#fs-mod
[15] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#termAnnotation
[16] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#candidates

the module. Those include properties such as `similarity`, `matchQuality`, `matchSuitability`, `type` of the candidate, but the only required property is a pointer identifying the relevant source content span to which the translation suggestion applies. Apart from reusing core, the module is extensible by the Metadata Module and by custom extensions.

The [21] Glossary Module[17] is designed to provide locally relevant Terminology matches. But can be also used *vice versa* to store terminology identified by human or machine agents during the roundtrip. A mapping of XLIFF Core + Glossary Module to and form TBX Basic has been defined in [24]. This mapping is now being standardized also within OASIS XLIFF OMOS TC (the home of LIOM and JLIFF).

The [21] Metadata Module[18] is perhaps the most suitable for being ported to JSON and other non-markup-language serialization methods. While being a module that in fact strictly protects it's content, it's also an extensibility mechanism for implementers who don't want to extend via their own namespace based extensions. The metadata structure is optionally recursive and is allowed on all XLIFF Core static structural elements (`file`, `group`, `unit`). It doesn't specify a referencing mechanism. It is simply an optionally hierarchically structured and locally stored set of key-value pairs to hold custom metadata. Because the data strusture is restricted to key-value pairs it provides at least some limited interoperability for the custom data as all implementers should be capable of displaying structural elements related key-value pairs data. Each Metadata Module object (element) has an optional id property (attribute) that allows for addressing from outside the module, either globally or within the LIOM instance.

The [21] Resource Data Module is designed to provide native resource data either as context for the Translatable payload or as non-text based resource data to be modified along with the Translatable payload (if the resource data model is known and supported by the receiving agent), for instance GUI resources to be resized to fit the Translated UI strings. Resource data can be considered locally specific skeleton data and would be typically binary or generally of any media type. To tell receiving agents what type of data the module holds, there is a media type property (`mimeType` attribute).

The [21] Size and Length Restriction Module[19] is a powerful mechanism that allows for defining any sort of size or storage constraints, even multidimensional. IT specifies a standard code point based restriction profile as well as three standard storage size restriction profiles. It also gives guidance how to specify arbitrary size or shape restriction profiles, for instance to control fitting restrictions in com-

[17] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#glossary-module
[18] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#metadata_module
[19] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#size_restriction_module

plex desktop publishing environments, custom embedded displays with font limitations, and so on.

The [21] Validation Module[20] provides an end user friendly way to specify simple Quality Assurance rules that target strings need to fulfill, mainly in relation to source strings. For instance a substring that appears in source must or must not appear in the target string. For instance, a brand name must appear in both source and target. Or on the contrary, a brand name must not be used in a specific locale for legal reasons.

3.2.2. The ITS Module

The [21] ITS Module[21] specification is comparable in size with all the other Module specifications taken together. It defines native support or mapping, Extraction or ITS Rules parsing guidance for all 19 W3C ITS 2.0 [8] metadata categories and for its [8] ITS Tools Annotation[22] mechanism.

Language Information[23] uses the [5] data model via `xml:lang` to indicate the natural language of content. This is obviously very useful in case you want to source translations or even just render the content with proper locale specifics. This partially overlaps with XLIFF's own usage of `xml:lang` to specify `srcLang` and `trgLang`. An `urn:oasis:names:tc:xliff:itsm:2.1` namespace based attribute `itsm:lang` is provided to specify third language material inline. Both xsd and JSON Schema have an issue in validating [5] tags. A regex or custom code based solution is recommended.

Directionality[24] has quite a profound Internationalization impact, it let's renderers decide at the protocol level (as opposed to the plain text or script level) whether the content is to be displayed left to right (LTR - Latin script default) or right to left (RTL - Arabic or Hebrew script default). But the Unicode Bidirectional Algorithm [17]as well as directionality provisions in HTML and many XML vocabularies changed since 2012/2013, so the ITS 2.0 specification text is actually not very helpful here. This obviously doesn't affect the importance of the abstract data category and of having proper display behavior for bidirectional content. LIOM contains core `srcDir` and trgDir and `dir` properties that allow values `ltr`, `rtl`, and `auto`. The default auto determines directionality heuristically as specified in [17]. Directionality in XLIFF is given by a high level protocol in the sense of [17]. All objects that can have the directionality property in LIOM either determine the directionality of their descendants (as higher level protocol) or act as directionality isolators inline.

[20] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#validation_module
[21] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#ITS-module
[22] https://www.w3.org/TR/its20/#its-tool-annotation
[23] http://www.w3.org/TR/its20/#language-information
[24] http://www.w3.org/TR/its20/#directionality

Preserve Space[25] indicates via `xml:space` whether or not whitespace characters are significant. If whitespace is significant in source content it is usually significant also in the target content, this is more often then not an internal property of the content format, but it's important to keep this characteristics through transformation pipelines. The danger that this category is trying to prevent is the loss of significant whitespace characters that could not be recovered. This data category is considered XMLism or SGMLism. It should be preserved at LIOM level. However, even XLIFF21 recommends pre-Extaction normalization of whitespace and setting of all inline content whitespace behavior to `preserve`. This is also the option best interoperable with JSON where all payload whitespace is significant.

ID Value[26] indicates via `xml:id` a globally unique identifier that should be preserved during translation and localization transformations mainly for the purposes of reimport of target content to all the right places in the native environment. XLIFF id mechanism is NMTOKEN rather than NCName based. However, usage of `xml:id` to encode this metadata category can only be exprected in XML source environments. Therefore, XLIFF and LIOM use an unrestricted string mechanism (`original` on file, `name` on group and unit) to roundtrip native IDs.

Terminology[27] can simply indicate words or multi-word expressions as terms or non-terms. This is how the category worked in [7]. In [8], Terminology can be more useful by pointing to definitions or indicating a confidence score, which is especially useful in cases the Terminology entry was seeded automatically. Terminology belong exclusively to categories that come from the native format. Together with Text Analysis it can be actually injected into the content during any stage of the lifecycle or roundtrip and is not limited to source. However, it is very important for the localization process, human or machine driven, to have Terminology annotated be it even only the simple Boolean flag. Core [14] doesn't have the capability to say that a content span is not a term, therefore the negative annotation capability is provided via the ITS Module.

Text Analysis[28] is a sister category to Terminology that is new in [8]. It is intended to hold mostly automatically sourced (possibly semi-supervised) entity disambiguation information. This can be useful for translators and reviewers but can also enrich reading experience in purely monolingual settings. This is fully defined in the ITS Module as there is no equivalent in core [14] or [21].

Domain[29] can be used to indicate content topic, specialization or subject matter focus that is required to produce certain translations. This can be for instance used to select a suitably specialized MT engine, such as one trained on an automotive bilingual corpus in case an automotive domain is indicated or. In another

[25] http://www.w3.org/TR/its20/#preservespace
[26] http://www.w3.org/TR/its20/#idvalue
[27] http://www.w3.org/TR/its20/#terminology
[28] http://www.w3.org/TR/its20/#textanalysis
[29] https://www.w3.org/TR/its20/#domain

use case, a language service provider will use a sworn translator and require in country legal subject matter review in case the domain was indicated as legal. Although ITS data categories are defined independently and don't have implementation dependencies, Domain information is well suited for usage together with the Terminology and Text Analysis datacats. As the Domain[30] datacat doesn't have local markup in the W3C [8] namespace, [21] had to define a local `itsm:domain` attribute that is also taken over as a local property by [14] and [11].

MT Confidence[31], Localization Quality Issue[32], Localization Quality Rating[33], and Provenance[34] - all new categories in ITS 2.0 - can be only produced during Localization transformations; specifically, during Machine Translation, during a review or Quality Assurance process, during or immediately after a manual or automated Translation or revision.

MT Confidence[35] gives a simple score between 0 and 1 that encodes the automated translation system's internal confidence that the produced translation is correct. This score isn't interoperable but can be used in single engine scenarios for instance to color code the translations for readers or post-editors. It can also be used for storing the data for several engines and running comparative studies to make the score interoperable first in specific environments and later on maybe generally. This overlaps with the LIOM Translation Module's property `matchQuality`.

Localization Quality Issue[36] contains a taxonomy of possible Translation and Localization errors that can be applied in annotations of arbitrary content spans. The taxonomy ensures that this information can be exchanged among various Localization roundtrip agents. Although this mark up is typically introduced in a Bitext environment on target spans, marking up source isn't exclude and can be very practical, especially when implementing the feedback or even reporting a source issue. Importantly, the issues and their descriptions can be Extracted into target content and consumed by monolingual reviewers in the native environment. This is fully defined in the ITS Module as there is no equivalent in core [14] or [21].

Localization Quality Rating[37] is again a simple score that gives a percentage indicating the quality of any portion of content. This score is obviously only interoperable within an indicated Localization Quality Rating system or metrics. Typically flawless quality is considered 100 % and various issue rates per translated

[30] https://www.w3.org/TR/its20/#domain
[31] http://www.w3.org/TR/its20/#mtconfidence
[32] http://www.w3.org/TR/its20/#lqissue
[33] http://www.w3.org/TR/its20/#lqrating
[34] http://www.w3.org/TR/its20/#provenance
[35] http://www.w3.org/TR/its20/#mtconfidence
[36] http://www.w3.org/TR/its20/#lqissue
[37] http://www.w3.org/TR/its20/#lqrating

volume would strike down percentages, possibly dropping under an acceptance threshold that can be also specified. This is fully defined in the ITS Module as there is no equivalent in core [14] or [21].

Provenance[38] in ITS is strictly specialized to indicate only translation and revision agents. Agents can be organizations, people or tools or described by combinations of those. For instance, Provenance can indicate that the Reviser John Doe from ACME Language Quality Assurance Inc. produced a content revision with the Perfect Cloud Revision Tool. This is fully defined in the ITS Module as there is no equivalent in core [14] or [21].

In spite of [21] using the W3C namespace for the ITS Module, there is a systematic scope mismatch between the XLIFF defined ITS attributes and the ITS defined XML attributes. Because [8] has no provision to parse pseudo-spans, it will necessarily fail to identify spans formed by XLIFF Core `<sm/>` and `<em/>` markers.

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

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

4. The design of JLIFF

The design of JLIFF follows the design of XLIFF 2 closely albeit making use of abstractions described under LIOM Core. As with XLIFF 2, the JLIFF Core corresponds to the abstract Localization Interchange Object Model, LIOM. One of the primary goals of JLIFF is compatibility with XLIFF 2 to allow switching between XML and JSON based pipeline at will as stated in the Introduction. While JLIFF is structurally different compared to XLIFF 2 due to the much simpler JSON representation format, compatibility is made possible through the following mappings:

[38] http://www.w3.org/TR/its20/#provenance

1. As a general rule, JSON object property names are used to represent XLIFF elements and attributes, with the exception of element sequences that must be represented by JSON arrays. JSON object properties should be unique and are unordered, whereas this does not generally hold for XML elements;
2. It was decided to use JSON arrays to represent element sequences, for example a sequence of `<file>` elements becomes an array identified by the JSON object property `"files": […]` where each array item is an anonymous file object that contains an array of `"subfiles": […]`. It was decided to use plural forms to refer to arrays in JLIFF and as a reminder of the structural differences between XML and JSON;
3. To store units and groups that exist within files, JSON object property `"subfiles": […]` is an array of unit and group objects representing XLIFF `<unit>` and `<group>` elements, where an anonymous unit object is identified by a `"kind": "unit"` and an anonymous group object is identified by `"kind": "group"`;
4. Likewise, `"subunits": […]` is an array of subunits of a unit object, where a segment subunit is identified as an object with `"kind": "segment"` and a ignorable object is identified as `"kind": "ignorable"`;
5. A subset of XSD data types that are used in XLIFF are also adopted in the JLIFF schema by defining corresponding JSON schema string types with restricted value spaces defined by regex patterns for `NCName`, `NMTOKEN`, `NMTOKENS`, and `URI/IRI`. The latter is not restricted by a regex pattern due to the lax validation of URI and IRI values by processors;
6. Because JSON intrinsically lacks namespace support, it was decided to use qualified JSON object property names to represent XLIFF modules, which is purely syntactic to enhance JLIFF document readability and processing. For example, ITS module properties are identified by prefix `its_`, such as `"its_locQualityIssues"`. Generally undersore "_" is used as the namespace prefix separator for modules (unlike custom namespace based extentions);
7. JLIFF extensions are defined by the optional JSON-LD context `"@context": {…}` as a property of the anonymous JLIFF root object. [12] offers a suitable replacement of XML namespaces required for extension identification and processing. A JSON-LD context is a mapping of prefixes to IRIs. A JSON-LD processor resolves the prefix in an object property name and thus creates a fully qualified name containing the corresponding IRI qualifier;
8. To identify JLIFF documents, the anonymous JLIFF root object has a required property `"jliff": "2.0"` or `"jliff": "2.1"`;
9. One of the decisions taken in relation to element mappings was not to explicitly support well-formed `<pc/>` and `<mrk/>` elements, therefore `<mrk/>` is

mapped to <sm/> and <em/> pairs, and <pc/> is mapped to <sc/> and <ec/> pairs. See also LIOM Core.

These mapping decisions were verified against the LIOM and XLIFF 2 while developing the JSON schema for JLIFF. In addition, to validate the approach several XLIFF examples were translated to JLIFF and validated by the JLIFF JSON schema. A reference implementation is being developed for lossless translation of XLIFF into JLIFF and back, except for support for <pc/> and <mrk/> elements as explained above.

Further design considerations worth noting include:

1. While JSON supports the Boolean type values `true` and `false`, it was decided to use string based enumerations of `yes` and `no` in JLIFF to represent XLIFF attributes of the `yesNo` type. There are two main reasons for this. Firstly, the omission of a Boolean value is usually associated with the value false by processors and applications. By contrast, the absence of a value should not default to false in JLIFF. The absence of a value has an XLIFF driven meaning. In fact the XLIFF default of the `yesNo` attributes is `yes`. Absence of of an attribute typically indicates permission. Hence we defined the yes defaults in the JLIFF JSON schema, which would have conflicted with the defaulting behavior of the JSON Boolean type. Secondly, the object property `canReorder` of the `ec`, `ph`, and `sc` objects is a three-valued enumeration with the `yes`, `no`, and `firstNo` values, necessitating the use of a JSON string type with enumeration rather than a JSON Boolean type in JLIFF.
2. Almost all JSON schema types defined for JLIFF correspond one-to-one with JSON object property names in JLIFF documents. This design choice reduces efforts to comprehend the JLIFF JSON schema structure for implementers versed in XLIFF. For example, the `files` property mentioned earlier has a corresponding `files` type in the JLIFF JSON schema, which is an array that references the `file` schema type. However, this schema design deviates in one important aspect that is intended to avoid unnecessary duplication of the schema types for the properties `subfiles` and `subgroups` that share the same data model. It was decided to introduce the schema type `subitems` to represent the value space of both `subfiles` and `subgroups`. We also added named types to the schema that have no corresponding property name, to break out the JSON structure more clearly. For example, `elements` is an array of mixed types, which is `one of` (`element-text`, `element-ph`, `element-sc`, `element-ec`, `element-sm`, and `element-em` in the schema. Note that `element-text` is a string while the other types are objects.
3. JLIFF considers LIOM Modules an integral part of the JLIFF specification, meaning that all Modules are part of the single JSON schema specification of JLIFF [11]. The decision to make Modules part of the JLIFF specification is a practical one that simplifies the processing of Modules by processors, as Mod-

ules are frequently used (albeit different subsets based on individual needs and specializations) by implementers. By contrast, extensions are registered externally and included in JLIFF documents as `userdata` objects. A `userdata` object contains one or more extensions as key-value pairs: each extension is identified by a qualified property with an extension-specific JSON value. The prefix of the qualified property of an extension is bound to the IRI of the extension using the JSON-LD `@context` to identify the unique extension namespace IRI. Processors that are required to handle extensions should resolve the prefix to the extension's objects fully qualified names as per JSON-LD processing requirements. Otherwise extensions can be ignored without raising validation failures. This approach offers an extensible and flexible mechanism for JLIFF extensions. While the JSON-LD workaround for namespaces was considered a suitable solution for extensions, it was considered heavy weight and to complex for modules that are used regularly. The "context" of modules is considered a shared JLIFF agent knowledge documented in the prose specification rather then being resolved each time when module data need processed, hammering OASIS servers that would need to hold the canonical context files required for proper JSON-LD processing.

5. Reference Implementation

It was an early goal to work on a concrete implementation of JLIFF in parallel to the development of the schema. It would give us an early opportunity to find and work through any design flaws or limitations. Fortunately, the JliffGraphTools [9] reference implementation has been open source since early on.

It was a wish that JLIFF should be easy to implement and serialize/deserialize using well-known JSON libraries.

As explained in the JLIFF Design section, there is a difference in the structure of JLIFF and XLIFF in inline markup. In XLIFF, inline markup is nested within the segment text as descendant elements of the `<source>` and `<target>` elements. In JLIFF the segment text and inline markup are stored as an array of objects property of the unit's source and target properties. This has an impact on how rendering tools may display strings for translation, see below on the approach taken in [9].

Having got the priority of JSON serialization and deserialization working we then looked at roundtripping, i.e. the capability to create an XLIFF output from a JLIFF file legally changed by a Translation agent. JliffGraphTools [9] supports bidirectional serialization between XLIFF and JLIFF and it is this library which powers the Xliff2JliffWeb web application made public at http://xliff2jliff.azurewebsites.net/. Unfortunately the public web application only implements the JLIFF output capability at the time of writing this.

At present when segments for translation are rendered in [9], there is an option to flatten the array of text and inline markup objects and render them in a way which is based upon the approach taken in the [16] XLIFF library. That is inline markup tags are converted to coded text which uses characters from the private use area of [18] to delimit inline markup tags. See [16] http://okapiframework.org/devguide/gettingstarted.html#textUnits.

The following program listings demonstrate [9] can be used to exchange the flattened fragments instead of the fully equivalent JLIFF. This capability is based on JLIFF having been designed to support any of the four logically possible LIOM roots, see LIOM Core Structure.

Example 14. Simple XLIFF input

```
<?xml version="1.0"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:2.0" version="2.1"
srcLang="en" trgLang="fr">
<file id="f1">
 <unit id="u1">
  <originalData>
   <data id="d1">[C1/]</data>
   <data id="d2">[C2]</data>
   <data id="d3">[/C2]</data>
  </originalData>
  &lt;segment canResegment="no" state="translated">
   <source><ph id="c1" dataRef="d1"/> aaa <pc id="c2" dataRefEnd="d3"
dataRefStart="d2">text</pc></source>
   <target><ph id="c1" dataRef="d1"/> AAA <pc id="c2" dataRefEnd="d3"
dataRefStart="d2">TEXT</pc></target>
  &lt;/segment&gt;
  <ignorable>
   <source>.  </source>
  </ignorable>
 </unit>
</file>
</xliff>
```

Example 15. [XLiff2JliffWeb] -> Fully equivalent JLIFF

```
{
  "jliff": "2.1",
  "srcLang": "en-US",
  "trgLang": "fr-FR",
  "files": [
    {
      "id": "f1",
      "kind": "file",
```

```
"subfiles": [
  {
    "canResegment": "no",
    "id": "u1",
    "kind": "unit",
    "subunits": [
      {
        "canResegment": "no",
        "kind": "segment",
        "source": [
          {
            "dataRef": "d1",
            "id": "c1",
            "kind": "ph"
          },
          {
            "text": " aaa "
          },
          {
            "dataRef": "d3",
            "id": "c2",
            "kind": "ec"
          },
          {
            "text": "text"
          },
          {
            "kind": "ec"
          }
        ],
        "target": [
          {
            "dataRef": "d1",
            "id": "c1",
            "kind": "ph"
          },
          {
            "text": " AAA "
          },
          {
            "dataRef": "d3",
            "id": "c2",
            "kind": "ec"
          },
          {
            "text": "TEXT"
```

```
                },
                {
                  "kind": "ec"
                }
              ]
            },
            {
              "kind": "ignorable",
              "source": [
                {
                  "text": ".  "
                }
              ],
              "target": []
            }
          ]
        }
      ]
    }
  ]
}
```

Example 16. [XLiff2JliffWeb] -> "Flattened" JLIFF

```
{
  "jliff": "2.1",
  "srcLang": "en",
  "trgLang": "fr",
  "subunits": [
    {
      "canResegment": "no",
      "kind": "segment",
      "source": [
        {
          "dataRef": "d1",
          "id": "c1",
          "kind": "ph"
        },
        {
          "text": " aaa "
        },
        {
          "dataRef": "d3",
          "id": "c2",
          "kind": "ec"
        },
        {
```

```
            "text": "text"
          },
          {
            "kind": "ec"
          }
        ],
        "target": [
          {
            "dataRef": "d1",
            "id": "c1",
            "kind": "ph"
          },
          {
            "text": " AAA "
          },
          {
            "dataRef": "d3",
            "id": "c2",
            "kind": "ec"
          },
          {
            "text": "TEXT"
          },
          {
            "kind": "ec"
          }
        ]
      }
    ]
  }
```


6. Discussion and Conclusions

In the above we tried to show how we ported into JSON XLIFF 2, a complex business vocabulary from the area of Translation and Localization. While the paper deals in detail only with XLIFF and JLIFF, we believe that important topics were covered that will be useful for designer who will endeavor porting their own specialized multi-namespace business vocabularies into JSON.

The major takeway we'd like to suggest is not to port directly from XML to JSON. It is worth the time to start your exercise with expressing your XML data model in an abstract way, we used UML class diagram as the serialization independent abstract method.

XML serializations are as a rule fraught with "XMLism" or "SGMLism". While some of the XML capabilities such as the multi-namespace support are clear XML advantages that will force the JSON-equivalent-designer into complex and more

or less elegant workarounds and compromises. Some other XML traits and constraints are arbitrary from the point of view of other languages and serialization methods.

To name just a few examples of XMLism that doesn't need maintained in JSON. You don't need to support well formed versions of inline markup in JSON, it is easier to serialize everything linearly. In JSON, all payload space is significant, so you don't need to keep the `preserve | default` flag in your JSON serialization. Instead make sure that all inline data is normalized and set to `preserve` in your XML data. JSON data types are much poorer than XML datatyes, nevertheless, you can make up for this with relative ease with the usage of regular expression patterns in your JSON schema. For instance

Example 17. NCName pattern in JSON schema

```
"NCName": {
      "description": "XSD NCName type for xml:id interoperability",
      "type": "string",
      "pattern": "^[_A-Za-z][-._A-Za-z0-9]*$"
    }
```

Namespaces support workarounds in JSON are worth an extra mention. While JSON doesn't support namespaces *per se*. We identified the JSON-LD methods for introducing and shortening fully qualified names quite useful as a namespaces support surrogate. For practical reasons (like prevention of hammering of OASIS servers to read XLIFF module context files) we decided to use the full blown JSON-LD method for expanding prefixes into fully qualified names only for extensions. We decided to use an arbitrary "_" (underscore) prefix separator to make the XLIFF modules human discernable. There goes the disadvantage of losing the modularity of XLIFF modules in JLIFF, yet we felt that JSON-LD-coding of each of the modules data would be very complex and heavyweight with minor benefits to outweigh the drawbacks.

Bibliography

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

[2] S. Saadatfar and D. Filip, *Best Practice for DSDL-based Validation*. in XML London 2016 Conference Proceedings, May 2016. `https://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=64`

[3] D. Filip, *W3C ITS 2.0 in OASIS XLIFF 2.1*, in XML Prague 2017 - Conference Proceedings, Prague, 2017, vol. 2017, pp. 55–71. `http://`

archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=67

[4] S. Bradner and B. Leiba, Eds., *Key words for use in RFCs to Indicate Requirement Levels and Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, IETF (Internet Engineering Task Force) 1997 & 2017. http://tools.ietf.org/html/bcp14

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

[6] S. Faulkner at al. Eds., *HTML 5.2* W3C Recommendation 14 Dec 2017. https://www.w3.org/TR/html52/

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

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

[9] P. Ritchie, *JLIFF Graph Tools*. Vistatec, 2019. https://github.com/vistatec/JliffGraphTools/commit/74ffde990d8dd6d6d5d3f80d78e76ea8b0dc8736

[10] D. Filip and R. van Engelen, *JLIFF Version 1.0 [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-jliff/commit/7e63e0d766bb7394f9dccaa93d7fa54bf1a394d3

[11] R. van Engelen, *JLIFF Version 1.0, JSON Schema [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-jliff/commit/2ed3b57f38548600f1261995c466499ad0ade224/>

[12] M. Sporny, G. Kellogg, M. Lanthaler, Eds. *JSON-LD 1.0, A JSON-based Serialization for Linked Data* W3C Recommendation 16 January 2014. https://www.w3.org/TR/2014/REC-json-ld-20140116//>

[13] D. Filip: Localization Standards Reader 4.0 [v4.0.1], Multilingual, vol. 30, no. 1, pp. 59–73, Jan/Feb-2019. https://magazine.multilingual.com/issue/jan-feb-2019dm/localization-standards-reader-4-0/

[14] D. Filip, *XLIFF 2 Object Model Version 1.0 [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-om/commit/030828c327998e7c305d9be48d7dbe49c8ddf202/>

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

[16] Y. Savourel et al., *Okapi Framework*. Stable release M36, Okapi Framework contributors, August 2018. http://okapiframework.org/

[17] M. Davis, A. Lanin, and A. Glass, Eds.: *UAX #9: Unicode Bidirectional Algorithm.*. Version: Unicode 11.0.0, Revision 39, 09 May 2018. Unicode Consortium. http://www.unicode.org/reports/tr9/tr9-39.html

[18] K. Whistler et al., Eds.: *The Unicode Standard*. Version 11.0 - Core Specification, 05 June 2018. Unicode Consortium. https://www.unicode.org/versions/Unicode11.0.0/UnicodeStandard-11.0.pdf

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

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

[21] D. Filip, T. Comerford, S. Saadatfar, F. Sasaki, and Y. Savourel, Eds.: *XLIFF Version 2.1*. OASIS Standard, 13 February 2018. OASIS http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html

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

[23] D. Filip and J. Husarčík, Eds., *XLIFF 2 Extraction and Merging Best Practice, Version 1.0*. Globalization and Localization Association (GALA) TAPICC, 2018. https://galaglobal.github.io/TAPICC/T1/WG3/rs01/XLIFF-EM-BP-V1.0-rs01.xhtml/>

[24] J. Hayes, S. E. Wright, D. Filip, A. Melby, and D. Reineke, *Interoperability of XLIFF 2.0 Glossary Module and TBX-Basic, Localisation Focus*, vol. 14, no. 1, pp. 43–50, Apr. 2015. https://www.localisation.ie/resources/publications/2015/260

[25] D. Filip and J. Husarčík, *Modification and Rendering in Context of a Comprehensive Standards Based L10n Architecture*, Proceedings ASLING Translating and the Computer, vol. 40, pp. 95–112, Nov. 2018. https://www.asling.org/tc40/wp-content/uploads/TC40-Proceedings.pdf

History and the Future of Markup

Michael Piotrowski
Université de Lausanne, Section des sciences du langage et de l'information
<michael.piotrowski@unil.ch>

1. Introduction

The report of XML's death has been greatly exaggerated, but it is becoming obvious that the halcyon days are over. To be sure, XML has enjoyed tremendous success since its first publication as a W3C Recommendation in 1998. Nowadays there are few areas of computing where, in some way or another, XML does not play a role. There are probably hundreds of specifications and standards built on XML, and dozens of related technologies, such as XSLT, XQuery, XPath, XML Schema, XLink, XPointer, XForms, etc. The W3C press release on the occasion of the tenth anniversary of XML quoted Tim Bray, one of the editors of the XML Recommendation, as saying, "[t]here is essentially no computer in the world, desk-top, hand-held, or back-room, that doesn't process XML sometimes."[1] This is certainly still true today: at the height of the hype, XML found its way into so many applications, from configuration files to office documents, it is unlikely to completely disappear anytime soon—even though it may become legacy technology.

Nevertheless, when it comes to formats for representing and exchanging structured data, JSON is now all the rage;[2] for narrative documents, Markdown enjoys a similar role. The W3C's XML working groups have all been closed, and HTML development has been taken over by what is essentially an industry consortium that opposed the transition of HTML to XML. The primary syntax of HTML5 *looks* like SGML, but the specification explicitly states that HTML is *not* an SGML application: "While the HTML syntax described in this specification bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules."[3]

Markdown and similar lightweight markup languages clearly offer writers a much more compact syntax for authoring simple documents, but even slightly more complex documents require extensions. Consequently, people have defined a large number of mutually incompatible extensions for different purposes. Pandoc[4] does an amazing job at integrating many of them into a useful whole; one is reminded of this 1989 speculation about the future:

[1] W3C XML is Ten! [http://www.w3.org/2008/xml10/xml10-pressrelease]
[2] Sinclair Target, "The Rise and Rise of JSON [https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html]"
[3] HTML Standard [https://html.spec.whatwg.org/#parsing]

> *A new generation of software products may change this: perhaps the most desirable result would be that by virtue of the markup minimization capability, together with smart editors, authors will be using SGML without knowing about it, whilst publishers reap the benefits. [1]*

Except for the SGML part, of course: none of this is formally standardized.[5]

JSON and Markdown are in some respects certainly more convenient than XML, but hardly "better" in an absolute sense, in particular not from a computer science perspective. By defining an SGML DTD for HTML 5.1, Marcus Reichardt has demonstrated that, "while nominally not based on SGML, owing to HTML requiring legacy compatibility, HTML5 hasn't striven far from its SGML roots, containing numerous characteristics traceable to SGML idiosyncrasies." [20]. This, as well as the bitter conflicts between the W3C and WHATWG,[6] also suggests that the dissociation of HTML5 from XML and SGML is not due to technical requirements. It rather goes to show that the development of technology is not solely determined by the technical superiority of one system over another, but that it is to a large extent also driven by cultural forces, fads, and fashions. As technology is created and used by humans, it is a cultural artifact.

On the one hand, it is thus more or less unavoidable that preferences change for apparently "no good reason," or that small practical advantages in one area are used to justify giving up important benefits in other areas. On the other hand, given the investments made into the creation of a complex ecosystem such as that of XML, it would be a shame to simply throw it all away. This obviously applies to business investments, but what is much more important is the *intellectual* investment, the experiences and insights gained in the process.

2. Why we Need a History of Markup

This is why we need history of technology, in this case: a history of markup technology. If we want to advance the field rather than reinvent the wheel, we need to know by which ways we arrived at the point where we are now—including the roads *not* taken. Software, including markup languages, file formats, etc., are a very peculiar class of artifacts: as they are not governed by the laws of physics, designers enjoy, for better or worse, almost unlimited flexibility.

---

[4] Pandoc: a universal document converter [https://pandoc.org/]

[5] The CommonMark [https://commonmark.org/] initiative is working on a "standard, unambiguous syntax specification for Markdown, along with a suite of comprehensive tests to validate Markdown implementations against this specification."

[6] The conflict is even evident at many points in the HTML Standard [https://html.spec.whatwg.org/multipage/parsing.html#parsing]; for example, it—correctly—notes that "few (if any) web browsers ever implemented true SGML parsing for HTML documents" and that "the only user agents to strictly handle HTML as an SGML application have historically been validators." Claiming that this "has wasted decades of productivity" and that HTML5 "thus returns to a non-SGML basis" however, can only be interpreted as a dig at the W3C.

There are of course papers that take a historical perspective on markup and related technologies. For example, noting that "[d]ocument preparation has been an increasingly important application of computers for over twenty-five years," Furuta set out in 1992 to "identify those projects that have been especially influential on the thinking of the community of researchers who have investigated these systems" [5]. However, this overview (which covers publications up to 1988) was not intended as a *history* of document preparation, i.e., it does not link developments and suggest causalities. An actual history of markup would be a topic for at least one PhD thesis. The goal of this paper is thus merely to encourage a reflection on the history of markup, using SGML and XML as an example.

3. The Historical Trajectory of SGML and XML

As is well known, XML is an evolution of SGML, or, in the words of Egyedi and Loeffen [4], XML was "grafted" onto SGML. It thus has a clearly identifiable direct historical predecessor. The history of SGML has been documented several times by its "father," Charles Goldfarb, for example in Appendix A of The SGML Handbook, "A Brief History of the Development of SGML" [8]. SGML is an evolution of GML, a set of macros for IBM's SCRIPT formatter (itself modeled on Jerry Saltzer's RUNOFF) inspired by the idea of *generic coding*, which emerged in the late 1960s. Generic coding describes the idea of marking up text elements for their function (e.g., "heading") rather than their appearance (e.g., "Helvetica Bold 14/16, centered"). Generic coding thus introduced an abstraction and advanced the separation of content and form. Invented around the same time, Stanley Rice's *text format models* [21] can be considered the counterpart of generic coding in that it permits designers to systematically map these abstract structures to concrete formatting.[7] Taking these ideas together, one could thus mark up text as "heading" and then independently specify that headings are to be mapped to the model "LDa" (14-point sans serif bold) for one application or "LBd" (12-point text bold italic) for another—or the information "heading" could be used for the purpose of information retrieval. It is not hard to see that these ideas were very appealing for applications such as legal publishing, where highly structured texts are to be formatted in various ways for print and ideally also made available in electronic form, and thus also for IBM [7].

Goldfarb then went on to lead the development and standardization of GML into SGML, published as an international standard in 1986 [10]. In "The Roots of SGML—A Personal Recollection,"[8] he writes:

[7] In fact, Goldfarb has stated that "Stanley Rice, then a New York book designer and now a California publishing consultant, provided my original inspiration for GML." [7]
[8] Charles F. Goldfarb, "The Roots of SGML—A Personal Recollection [http://www.sgmlsource.com/history/roots.htm]"

> *After the completion of GML, I continued my research on document structures, creating additional concepts, such as short references, link processes, and concurrent document types, that were not part of GML. By far the most important of these was the concept of a validating parser that could read a document type definition and check the accuracy of markup, without going to the expense of actually processing a document. At that point SGML was born – although it still had a lot of growing up to do.*

The history of XML obviously does not begin with the publication of the W3C XML 1.0 Recommendation in 1998, nor with the creation of the SGML Editorial Review Board (ERB) by the W3C, which developed it, but the Web rather represented an incentive to revisit earlier proposals for simplifying SGML, such as Sperberg-McQueen's "Poor-Folks SGML"[9] or Bos's "SGML-Lite"[10]—or, as DeRose put it, "XML stands within a long tradition of SGML simplification efforts" [3]. From today's perspective, this historical development appears completely logical, as it follows a familiar narrative: from humble beginnings to great success. However, it is important to recognize that this well-known narrative is just *one* of many possible narratives. It mentions neither alternative approaches nor criticism, nor failures. The Office Document Architecture (ODA, ISO 8613) [12] is all but forgotten today, but it is one example of a quite different contemporaneous approach to more or less the same goals as SGML.[11] Criticism of the very idea of embedded markup is far from new either; for example, Raymond et al. claimed in 1993 that since "the formal properties of document management systems should be based on mathematical models, markup is unlikely to provide a satisfactory basis for document management systems" [19]. Robin Cover has called this report "a reminder that SGML has had significant intelligent detractors from the beginning."[12] In fact, many of these lines of criticism continue until today.

4. Some Observations on SGML

At this point we would like to point out some historically interesting observations that—while far from being obscure—seem to be less often discussed than, for

[9] Michael Sperberg-McQueen, "PSGML: Poor-Folks SGML: A Subset of SGML for Use in Distributed Applications [http://www.tei-c.org/Vault/ED/edw36.gml]," Document TEI ED W 36, October 8, 1992.
[10] Bert Bos, "'SGML-Lite' – an easy to parse subset of SGML [https:// www.w3.org/ People/ Bos/ Stylesheets/SGML-Lite.html]," July 4, 1995.
[11] The Wikipedia article Open Document Architecture [https:// en.wikipedia.org/ wiki/ Open_Document_Architecture] (*Open Document Architecture* is the name used by ITU for their version of the otherwise identical standard) states: "It would be improper to call ODA anything but a failure, but its spirit clearly influenced latter-day document formats that were successful in gaining support from many document software developers and users. These include the already-mentioned HTML and CSS as well as XML and XSL leading up to OpenDocument and Office Open XML." Given the cleavage between SGML and ODA approaches and communities [15], we find this statement rather dubious without further support.
[12] http://xml.coverpages.org/markup-recon.html

example, the verbosity of the Concrete Reference Syntax (which is the only syntax for XML) or the problem of overlapping markup.

SGML is an extremely complex standard, and, as DeRose has remarked, the "list of SGML complexities that do not add substantial value is quite long" [3]. Some of these complexites are easy to explain historically. One example is markup minimization, which does not only include facilities for omitting start and end tags, but also *data tags*, text that functions both as content *and* as markup:[13] these features were motivated by the desire to minimize the amount of typing necessary when creating SGML documents with a simple text editor. Another example is the *character set description* in the SGML declaration, necessitated by the diversity of character sets and encodings in use at the time.

The reasons for other complexities are, however, less clear. For example, despite SGML's roots in a commercial product and extensive experience with it, many aspects necessary for interoperable implementations were left undefined, such as the resolution of public identifiers. Similarly, SGML does hardly say anything about documents may be processed apart from validation, in particular how they could be formatted for display or transformed for an information retrieval system. Macleod et al. criticized this in 1992 as follows:

> *SGML is a passive standard. That is, it provides mechanisms through which descriptive markup to be applied to documents but says nothing about how these documents are to be processed. The SGML standard refers frequently to the "application" but includes no clean mechanism for attaching applications to SGML parsers. [14]*

In fact, formatting seems to have been hardly a concern, as there is little in the standard related to formatting SGML documents or for interfacing with a formatter, and the facilities that are available—namely, link process definitions (LPD)—are not only weak, but also extremly complex, in particular in relation to what one can accomplish with them. In the commentary to Appendix D of the standard, entitled "LINK in a Nutshell," Goldfarb himself notes that "the problem is not so much putting LINK in a nutshell as keeping it there" [8]. This is well known—also because most of these features were removed from XML—but the question is why so much effort was expanded that quickly turned out to be of little use.

Another observation is that the SGML standard is strangely detached from computer science terminology and concepts that were already well-established at the time when the work on it started (between 1978 and 1980). Some of this is related to terminology, such as the use of the term *parser*: Barron noted in 1989 that the term *parser* is "firmly established in the SGML community" (in fact, it is

[13] In the SGML Handbook, Goldfarb calls data tags "to some extent an accident of history" and despite having been largely supplanted by short references, "still quite complex for both user and implementer" [8].

defined and used by the standard), whereas "a computer scientist would recognise the SGML processor as a parser generator or compiler-compiler which takes a formal specification of a language (the DTD) and generates a parser for that language, which in turn is used to process the user's document" [1].

Some problems are more serious; DeRose points out that "since SGML was developed with a publishing viewpoint largely divorced from computer science and formal language theory, it made some choices that led to bizarre consequences for implementers and users." [3] Kaelbling noted in 1990:

> *Since SGML is a language intended for computer-based systems, it is reasonable (in the absence of convincing argument to the contrary) that it should follow established conventions within the realm of computer science. By following accepted methods of notation and structural configuration, languages can be processed by existing tools following well-understood theories and techniques, thus offering considerable savings to development efforts. These savings are realized by automatically performing tedious, error-prone calculations correctly and by checking and clarifying the formal descriptions of the languages. [13]*

In their 1993 review of SGML, Nordin et al. made a similar statement:

> *From a software engineering viewpoint, it would make sense to use whatever tools are available to build SGML-based software. It is therefore of some interest to make sure that a standard does not inadvertently complicate the product development process.*
>
> *As specified, SGML does not cleanly map to either LL(1) or LALR(1)-type grammars. This has made it difficult to build SGML applications as the commonly used implementation tools have been difficult to apply. [16]*

Another formulation of this criticism: "It is not possible to construct a conforming SGML parser using well known compiler construction tools such a Lex and Yacc, since the language is context sensitive." [18] The demand that the "specification of a standard should reflect the state of the art" and that to this end, "the grammar specifying SGML should be rewritten to allow for automatic processing by common tools and techniques" [13]] seems altogether reasonable. Nordin et al. [16]] refer to Kaelbling [13] as well as further authors to underline this point; they also point to dissenting opinions (including Goldfarb's). This is another case of a problem that is well known, but again the question is: why? We suspect that the documents referenced by Nordin may be difficult to obtain now, but they could provide valuable insights on the rather striking fact that the editors of SGML, an international standard, were either unaware of or chose to ignore both research and proven practice in a relevant field.

A related observation is that even though it was clear to the editors of SGML that an SGML document describes a tree, SGML almost exclusively focused on the syntax. Goldfarb noted:

> *SGML can represent documents of arbitrary structure. It does so by modeling them as tree structures with additional connections between the nodes. This technique works well in practice because most conventional documents ar in fact tree structures, and because tree structures can easily be flattened out for representation as character sequences.*
>
> *Except for the terminal nodes, which are "data", each node in an SGML document tree is the root of a subtree, called an "element". The descendants of a node are the "content" of that element. [8]*

Despite that fact that parsing SGML documents shares many commonalities with parsing programming language code—and this is also mentioned in the standard—the parallels between the tree represented by an SGML document and the abstract syntax tree (AST) produced by a parser for a programming language seem not to have been apparent for the longest time. Considering an SGML or XML document as a serialization of a tree that exists independently of the concrete serialization (rather than the other way round) is a very powerful notion, especially in conjunction with a standard API to manipulate the tree.

In hindsight this appears obvious, but historically it seems to be a realization that was apparently not obvious at all. This does not mean that nobody had thought of it. For example, Furuta and Stotts already noted in 1988 that one of the most important problems to be solved in document processing is "to determine how composite document objects may be converted from one structure to another" [6]] and presented a system for transforming document trees based on H-graphs [17], explictly mentioning SGML as a possible application. However, only with the Document Object Model (DOM) and XPath the idea that an XML document describes a tree that is *independent* of its serialization, and on which programs can operate, took hold and eventually became explicit.[14] It is this notion that we think is the real foundation of today's XML ecosystem.[15]

5. The Future of Markup

Price noted in 1998 that the "historical origins of SGML as a technique for adding marks to texts has left a legacy of complexities and difficulties which hinder its wide acceptance." [18] This was proven true by XML: it is fascinating how quickly it was adopted and how an extremely rich ecosystem developed once many of these complexities had been discarded.

The early adoption by US Department of Defense and other government agencies (even before the publication of the final standard) was probably the decisive

[14] Conceptually this notion obviously already existed in DSSSL [11], but it remains for the most part implicit. For example, the standard talks about "transforming one or more SGML documents into zero or more other SGML documents" (page 9), i.e., it is the serialized documents that are considered primary.

[15] And which allows XQuery, for example, to process JSON just like XML.

for SGML to survive despite all of its problems. Looking back, it seems that a closer link of SGML to computer science research would have made it much easier to create tools and thus promoted a wider adoption. It may also have permitted people to realize earlier that the abstract tree structure is more important than the concrete syntax;[16] this would have greatly reduced the importance attributed to syntax.

It is also interesting to see that markup minimization—arguably the most syntax-focused feature of SGML there is—was a central concern. Due to the problems it created, it was completely rejected by XML—the XML Recommendation famously states as design goal 10: "Terseness in XML markup is of minimal importance."[17] Apart from the necessity to abandon markup minimization to make DTD-less parsing possible, one could say that the focus on markup syntax had become obsolete by the time it was realized that it is the abstract tree that is important. However, it probably also caused the backlash that we can now observe in the rise of JSON, Markdown, and similar formats that are essentially minimized HTML, as well as the reintroduction of minimization into HTML5.

Already in 1994, a critic pointed out that "SGML relies on technology from the 1970s, when almost all computers were mainframes, almost all data was textual, and sharing data—much less applications—between hardware platforms was almost unheard of" [9]; Price noted in 1998 that "SGML 86 reflects thinking from the 1960's and 1970's. Texts were not supposed to vary while being read" [18]. In the last 20 years both the types of content and users' interaction with it have significantly changed: at least on the Web the assumption that a document is static data that is parsed and rendered in what essentially amounts to batch mode is no longer true. This becomes evident in the HTML5 specification:[18]

> *This specification defines an abstract language for describing documents and applications, and some APIs for interacting with in-memory representations of resources that use this language.*
>
> *The in-memory representation is known as "DOM HTML", or "the DOM" for short.*
>
> *There are various concrete syntaxes that can be used to transmit resources that use this abstract language, two of which are defined in this specification.*

[16] In this context, Eliot Kimbers reflection in "Monastic SGML: 20 Years On [http://drmacros-xml-rants.blogspot.com/2013/08/monastic-sgml-20-years-on.html]" are interesting: "As I further developed my understanding of abstractions of data as distinct from their syntactic representations, I realized that the syntax to a large degree doesn't matter, and that our concerns were somewhat unwarranted because once you parse the SGML initially, you have a normalized abstract representation that largely transcends the syntax. If you can then store and manage the content in terms of the abstraction, the original syntax doesn't matter too much."

[17] Extensible Markup Language (XML) 1.0 (Fifth Edition) [https://www.w3.org/TR/REC-xml/#sec-origin-goals]

[18] HTML Standard [https://html.spec.whatwg.org/multipage/introduction.html#html-vs-xhtml]

These three sentences alone reflect significant conceptual differences to SGML, that are not evident from the seemingly conservative syntax. These include the formulation "documents and applications," the central role the DOM and the APIs to manipulate it, and the decoupling of the DOM from a concrete syntax. When XML was introduced, a frequently mentioned advantage over HTML was the possibility for users to create their own elements—the semantics of which would have to be defined by some application. HTML5 introduces the notion of *custom elements*,[19] which superficially seems equivalent. In fact, however, both the elements *and* their behavior are specified programmatically (i.e., in JavaScript) through the DOM API by subclassing `HTMLElement` (or a subclass thereof). This is a very different approach: first, because the element is not defined on the markup level but on that of the DOM, and second, because it also directly associates semantics with the element.

While HTML documents have for quite some time been a mix of HTML and JavaScript code operating on the DOM, custom elements represent a significant conceptual shift: HTML documents now have definitely become programs, with markup just serving as "DOM literal" or a kind of "here-document" for JavaScript. Is this the future of markup? In any case, this is the direction HTML is taking.

6. Conclusion

XML has come a long way. Its development and that of its ecosystem into what we have today is the result of over 50 years of history of document processing. Some choices were made consciously, others less so, and some features can only be described as historical accidents. We must look back in order to understand what lies ahead of us; taking a historical perspective can help to uncover hidden assumptions, implicit notions, critical decisions, misunderstandings, and so on, which still shape our understanding today. For Despite their similar appearances, HTML5 is conceptually quite different from SGML and XML. On the other hand, despite its different appearance, Markdown is very close to the traditional processing model of SGML. Its growing popularity in more and more domains requires more and more extensions, but it lacks a well-defined extension mechanism. The CommonMark[20] initiative aims to define and standardize a single common vocabulary, whereas one of the fundamental assumptions of SGML and XML is that this is not possible.

Sperberg-McQueen said in 1992 that "part of its [SGML's] accomplishment is that by solving one set of problems, it has exposed a whole new set of problems."[21] This is particularly true in a historical perspective. The point of this paper is to encourage reflection on and discussion of the history of markup tech-

[19] HTML Standard [https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements]
[20] https://commonmark.org/

nologies to advance the state of the art. In order to recognize opportunities and limitations of new technologies it is necessary to be able to compare it to previous technologies; at the same time, the design of new markup languages (understood in a wide sense) is, like that of programming languages, “always subtly affected by unconscious biases and by historical precedent” [2]; even approaches that aim to be “radically new” define themselves with respect to what has been there before. Those who cannot remember the past are condemned to repeat it.

References

[1] David Barron 1989. Why use SGML. *Electronic Publishing*. 2, 1, 3–24.

[2] Michael F. Cowlishaw 1984. The design of the REXX language. *IBM Systems Journal*. 23, 4, 326–335. doi:10.1147/sj.234.0326.

[3] Steven J. DeRose 1999. XML and the TEI. *Computers and the Humanities*. 33, 1–2, 11–30.

[4] Tineke M. Egyedi and Arjan Loeffen 2002. Succession in standardization: Grafting XML onto SGML. *Computer Standards & Interfaces*. 24, 4, 279–290. doi:10.1016/s0920-5489(02)00006-5.

[5] Richard Furuta 1992. Important papers in the history of document preparation systems: Basic sources. *Electronic Publishing*. 5, 1, 19–44.

[6] Richard Furuta and P. David Stotts 1988. Specifying structured document transformations. *Document Manipulation and Typography. Proceedings of the International Conference*. Cambridge University Press, Cambridge, 109–120.

[7] Charles F. Goldfarb 1997. SGML: The reason why and the first published hint. *Journal of the American Society for Information Science*. 48, 7, 656–661. doi:10.1002/(sici)1097-4571(199707)48:7%3C656::aid-asi13%3E3.0.co;2-t.

[8] Charles F. Goldfarb 1990. *The SGML handbook*. Oxford University Press, Oxford, UK.

[9] George F. Hayhoe 1994. Strategy or SNAFU? The virtues and vulnerabilities of SGML. *IPCC 94 proceedings. Scaling new heights in technical communication*. IEEE, New York, NY, USA, 378–379.

[10] International Organization for Standardization 1986. *ISO 8879:1986. Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Geneva.

[21] Michael Sperberg-McQueen, “Back to the Frontiers and Edges. Closing Remarks at SGML ’92: the quiet revolution [http://www.w3.org/People/cmsmcq/1992/edw31.html], October 29, 1992.

[11] International Organization for Standardization 1996. *ISO/IEC 10179:1996. Information technology — Processing languages — Document Style Semantics and Specification Language (DSSSL)*. Geneva.

[12] Vania Joloboff 1986. Trends and standards in document representation. *Text Processing and Document Manipulation. Proceedings of the International Conference*. British Computer Society; Cambridge University Press, Cambridge, 107–124.

[13] Michael Kaelbling 1990. On improving SGML. *Electronic Publishing*. 3, 2, 93–98.

[14] Ian A. Macleod, Brent Nordin, David T. Barnard, and Doug Hamilton 1992. A framework for developing SGML applications. *Proceedings of Electronic Publishing 1992 (EP 92)*. Cambridge University Press, Cambridge, 53–63.

[15] Charles K. Nicholas and Lawrence A. Welsch 1992. On the interchangeability of SGML and ODA. *Electronic Publishing*. 5, 3, 105–130.

[16] Brent Nordin, David T. Barnard, and Ian A. Macleod 1993. A review of the Standard Generalized Markup Language (SGML). *Computer Standards & Interfaces*. 15, 1, 5–19. doi:10.1016/0920-5489(93)90024-l.

[17] Terrence W. Pratt 1983. Formal specification of software using H-graph semantics. *Graph-grammars and their application to computer science*. H. Ehrig, M. Nagl, and G. Rozenberg, eds. Springer. 314–332.

[18] Roger Price 1998. Beyond SGML. *Proceedings of the Third ACM Conference on Digital Libraries (DL '98)*. ACM Press, New York, NY, USA, 172–181.

[19] Darrell Raymond, Frank Tompa, and Derrick Wood 1993. *Markup reconsidered*. Technical Report #356. Department of Computer Science, The University of Western Ontario.

[20] Marcus Reichardt 2017. The HTML 5.1 DTD. *Proceedings of XML Prague 2017*. University of Economics, Prague, 101–118.

[21] Stanley Rice 1978. *Book design: Text format models*. Bowker, New York, NY, USA.

Splitting XML Documents at Milestone Elements Using the XSLT Upward Projection Method

Gerrit Imsieke
le-tex publishing services GmbH
<gerrit.imsieke@le-tex.de>

Abstract

Creating chunks out of a larger XML tree is easy if the splitting points correspond to natural structural units, such as chapters of a book. When the splitting points are buried at varying levels in deeply nested markup, however, the task becomes more difficult.

Examples for this problem include: Splitting mostly flat HTML at headings (that are sometimes wrapped in `divs` *or* `sections`*); splitting paragraphs at line breaks (even when some highlighting markup stretches across the line breaks); transforming tabulated lines (that also may contain markup around the tabs) into tables proper; splitting chapters and sections at page breaks, etc.*

The presented solution does this quite elegantly using XSLT 2.0 grouping and tunneled parameters.

It will be discussed how the performance of this method scales with input size, the number of split points, or chunk length; whether xsl:evaluate can be used to create a generic, configurable splitting method and whether the method works with streaming XSLT processing.

An interactive visualization of how this method works will be included in the presentation.

Keywords: XSLT, Grouping, XML Splitting, XML Chunking, Streaming, Publishing, Mixed Content

1. Introduction

From time to time people describe splitting/chunking problems on xsl-list and ask for solutions, and the author would reply with a description how he tackles this using the approach presented here. However, the abstract description has never led to people actually implementing the solution, therefore the author sometimes created the solutions for them. (He once even received a valuable bottle of scotch for it out of gratitude.)

Although this technique is just a bit of XSLT that needs to be adapted to each concrete situation, it is probably worth the while to make this solution more broadly known in the XML/XSLT community under its suggested name, "splitting by upward projection".

It is also interesting to discuss the related concept of a "scope" for operations such as whitespace normalization or splitting at line breaks. In typical document-centric markup languages such as DocBook, TEI, or JATS, a new scope is established by footnotes, table cells, or list items. A scope in this sense prevents, for example, a paragraph to be split erroneously at a line break that is part of an embedded footnote.

In addition, it will be discussed whether the XSLT 3.0 instruction `xsl:evaluate` can be used for making this splitting functionality more generic and which performance implications this might have.

Splitting large documents at arbitrarily deeply nested nodes is a scenario that has led people to wonder whether the upward projection method lends itself to streaming XSLT processing. It will be discussed whether this is possible at all.

2. Description

The upward projection method consists of the following XSLT matching templates and grouping instructions:

- Match a splitting root element.

  For chunking at headings in HTML or at page breaks in TEI, this is typically the `body` element. For splittings at line breaks, this is typically a `p` element.

- Select all leaf nodes.

  Typically, `select="node()[empty(node())]"`, but it can get more complex. For example, footnotes in a paragraph must be treated as an atomic leaf node. Otherwise, they will get split into two when they contain a line break.

- Group the leaf nodes.

  In the case of page breaks in TEI, this is done `group-starting-with="pb"`. It can get more convoluted though. In the case of `two-column-start` and `two-column-end` markers (an example will be given below, Section 3.2), there is a `for-each-group/ @group-ending-with` nested within the `for-each-group/ @group-starting-with`.

- For each group, process the splitting root in a dedicated XSLT mode (for ex., `split`) with the tunneled parameter `restricted-to` set to `current-group()/ ancestor-or-self::node()`.

  This upward-looking selection is where this method's name stems from.

- A "conditional identity template" that matches any node in this `split` mode will pick up the tunneled parameter and check whether the matching node is among the `$restricted-to` nodes.

  If it is contained in `$restricted-to`, it will be reproduced. Otherwise, nothing will be written to the result tree.

This method was first introduced by the author in 2010, as documented on an internal Wiki page (Figure 1). It has been used in many chunking tasks since then, but it has not found its way into an XSLT cookbook and there has not been any systematic performance analysis yet.

3. Examples

Upward projection is used in many splitting scenarios. Three of them have been presented by the author at XML Prague 2018 [1].

3.1. Split at Page Break

The solution was sketched in response to a question that was raised by Geert Bormans on Mulberry Technologies' XSLT mailing list [2].

In this example, we will use a TEI document that contains Martin Luther's 1522 translation of the New Testament into German [3]. Figure 2 shows part of the TEI code around a page break, including the breadcrumb path to this element.

The manuscript contains 452 `pb` elements at varying depths below `front` and `body`. The `pb` location paths and their frequencies are as follows:

```
<pbs>
   <pb path="/TEI/text/body/div/div/p/pb" count="238"/>
   <pb path="/TEI/text/body/div/div/pb" count="91"/>
   <pb path="/TEI/text/body/pb" count="52"/>
   <pb path="/TEI/text/body/div/pb" count="47"/>
   <pb path="/TEI/text/front/pb" count="11"/>
   <pb path="/TEI/text/body/div/p/pb" count="10"/>
   <pb path="/TEI/text/front/div/p/pb" count="3"/>
</pbs>
```

In terms of text length, chunk count, and varying splitting point depths, this document serves as a presumably well-suited source for performance measurements in Section 4.

The XQuery script to generate this report, and all other programs and interactive demos of this paper/presentation, can be found at [4].

Initial Attempt

Here's a step-by-step guide to set it up. In order to be broadly applicable, also for people who don't use an XSLT 3 processor yet, we require that the solution work

EntwicklerSeiten/XsltSeiten/DesignPatterns/SegmentationUsingUpwardProjection

Illustration des Problems

```
   /---body-----\
 div   /| | \ \   \ \
 /\   / | |  |  \  \ \
 p p h1 p p div  h1 p p
           / \
          img p
```

soll an h1 gesplittet werden

```
 body    body        body
 div    /| | \       /|\
 /\   h1 p p div   h1 p p
 p p          / \
             img p
```

oder

```
<p>bla <b>hurz<tab/>foo</b> ax knax<tab/> foo bar</p>
→
<td>bla <b>hurz</b></td> <td><b>foo</b> ax knax</td> <td>foo bar</td>
```

Methode

- Vom lokalen Root-Element aus (kann ein z.B. ein Absatz sein, der HTML-Body oder ein ganzes [(typischerweise leere Elemente wie br sowie Textknoten) identifizieren. Die Spliting-Punkte müss
- Die Blattelemente gruppieren, group-starting- bzw. ending-with-Attribut sind die Splitting-Punkte

Figure 1. A company-internal Wiki page that describes the upward projection method

with XSLT 2.0 and 3.0. We will later see modifications of this solution that require 3.0 features.

We assume that the document is processed in `#default` mode and that `xpath-default-namespace="http://www.tei-c.org/ns/1.0"`.

```
• luther_septembertestament_1522.TEI-P5.xml ×
TEI  text  body  div  div  p  pb
16995      es.</p>
16996     <lb/>
16997     <p><note resp="#AH" type="editorial"><ref
16998         target="https://www.bibleserver.com/text/LUT/Johannes1,47">Johannes
16999       1,47</ref></note>Jhe&#x017F;us &#x017F;ahe Nathanael zu yhm komen/ vnd
17000      &#x017F;p&#xA75B;icht von yhm/ &#x017F;ihe/ <fw type="catch" place="bottom">Eyn
17001       rechter</fw><lb/>
17002      <pb facs="#f0145" n="[139]"/>
17003      <fw type="header" place="top">Sanct Johannes. LXVI.</fw><lb/> Eyn rechter
17004      J&#x017F;raheliter/ ynn wilchem keyn trug i&#x017F;t/ <note resp="#AH" type="editorial"
17005         ><ref target="https://www.bibleserver.com/text/LUT/Johannes1,48">Johannes
17006       1,48</ref></note>Nathanael <choice>
17007       <orig>&#x017F;p&#xA75B;i</orig>
```

Figure 2. TEI markup of Luther's New Testament translation

As soon as the transformation encounters a splitting root element, it will continue in `split` mode. In this mode, the tunneled parameter `$restricted-to` will determine whether the next-matching template, which is the identity template, will copy the current node to the result tree.

```
<xsl:template match="node() | @*" mode="#default split">
  <xsl:copy>
    <xsl:apply-templates select="@*, node()" mode="#current"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="node()" mode="split" priority="1">
  <xsl:param name="restricted-to" as="node()*" tunnel="yes"/>
  <xsl:if test="exists(. intersect $restricted-to)">
    <xsl:next-match/>
  </xsl:if>
</xsl:template>
```

The splitting root element, `/TEI/text` or `/TEI`, will be transformed in `split` mode for each chunk. If there are 12 chunks, the splitting root will be transformed 12 times. What goes into a chunk is determined by the `$restricted-to` tunneled parameter. It holds all nodes that go into a given chunk.

How do we calculate `$restricted-to` for each chunk? A chunk consists of all nodes below the splitting root and above the chunk's leaf nodes, plus the splitting root and the leaf nodes. (Leaf nodes are those without children, for example, text nodes. But see the note below.) The milestone element at which the split is about to occur must always be one of the leaf nodes, too.

Note: Extended Leaf Node Definition

For splitting purposes, "leaf nodes" are not necessarily nodes without children.

In other splitting scenarios where the splitting point is not an empty element, for example a `footnote` in a `phrase` in a `para` that is supposed to split the `para`, the splitting points must always be part of the "leaf nodes" even if they are not empty milestone elements. In such a scenario the XPath expression to select all leaf nodes has to be changed from `descendant::node()[empty(node())]` to, for example,

```
descendant::node()[exists(self::footnote)
                   or empty(ancestor::footnote | node())]
```

or, in XPath 3.1,

```
outermost(descendant::node()[not(has-children())] |
          descendant::footnote)
```

In yet other scenarios where splitting should be performed at substrings in text nodes, milestone elements have to be created first, using `xsl:analyze-string` or similar methods, in a preprocessing pass.

So we will use `xsl:for-each-group[@group-starting-with]` with all leaf nodes as the grouping population and the splitting points as the start nodes in order to determine the leaf nodes for each chunk. (`@group-ending-with` is equally possible; in the case of page breaks in TEI, the `pb` element may hold attributes that pertain to the page that follows the break, therefore `@group-starting-with` seems more adequate here.)

Once we have determined the leaf nodes of a future chunk, the splitting root and the nodes in between (vertically) can be selected by looking up the ancestor axis from the leaf nodes and intersecting it with the descendants of the splitting root element.

The grouping template looks like this:

```
<xsl:template match="/TEI" mode="#default">
  <xsl:variable name="leaves" as="node()+"
                select="descendant::node()[empty(node())]"/>
  <xsl:for-each-group select="$leaves" group-starting-with="pb[@facs]">
    <xsl:variable name="restriction" as="node()+"
      select="current-group()/ancestor-or-self::node()"/>
    <xsl:apply-templates select="/TEI" mode="split">
      <xsl:with-param name="restricted-to" tunnel="yes"
                      select="$restriction"/>
    </xsl:apply-templates>
  </xsl:for-each-group>
</xsl:template>
```

As said above, it does not matter much whether we use `/TEI` or `/TEI/text` in the template's `@match` attribute. The `$leaves` and `$restriction` variables would hold almost the same nodes in each case, and the chunks will be generated by processing `/TEI` in any case.

We will modify the template slightly in order to write each chunk into a new file of its own:

```
<xsl:template match="/TEI" mode="#default">
  <xsl:variable name="leaves" as="node()+"
              select="descendant::node()[empty(node())]"/>
  <xsl:for-each-group select="$leaves" group-starting-with="pb[@facs]">
    <xsl:variable name="restriction" as="node()+"
      select="current-group()/ancestor-or-self::node()
              union /TEI/teiHeader"/>
    <xsl:variable name="out-name" as="xs:string"
      select="concat('out/pb', substring(@facs, 2), '.xml')"/>
    <xsl:result-document href="{$out-name}">
      <xsl:apply-templates select="/TEI" mode="split">
        <xsl:with-param name="restricted-to" tunnel="yes"
                        select="$restriction"/>
      </xsl:apply-templates>
    </xsl:result-document>
  </xsl:for-each-group>
</xsl:template>
```

The resulting file for the splitting point depicted in Figure 2 looks (almost) like in Figure 3:

"Almost" means that in the actual results, `teiHeader` will be missing because it is not among the ancestors of the leaf nodes (except for the first chunk), therefore the conditional identity template will not write anything to the result tree. The situation is similar to splitting `/html`: The `head` element will be missing if not taken care for. We can address this by adding another template in `split` mode with a higher priority than the conditional identity template:

```
<xsl:template match="teiHeader" mode="split" priority="1.5">
  <xsl:copy-of select="."/>
</xsl:template>
```

We will return to this example when analyzing and improving the performance of this solution in Section 4 and when putting it into an `xsl:package` as a generic, configurable splitting template (Section 5).

```
• pbf0145.xml ×
TEI  text  body  div  div  p  pb
1   <?xml version="1.0" encoding="UTF-8"?>↵
2   <TEI xmlns="http://www.tei-c.org/ns/1.0">↵
3    <teiHeader>↵[243 lines]
247  <text>↵
248   <body>↵
249    <div xml:id="Joh" n="1">↵
250     <div xml:id="Joh.1" n="2">↵
251      <p><pb facs="#f0145" n="[139]"/>↵
252       <fw type="header" place="top">Sanct Johannes. LXVI.</fw><lb/> Eyn rechter Jſraheliter/↵
253       ynn wilchem keyn trug iſt/ <note resp="#AH" type="editorial"><ref↵
254         target="https://www.bibleserver.com/text/LUT/Johannes1,48">Johannes↵
255        1,48</ref></note>Nathanael <choice>↵
256        <orig>ſpꝛi</orig>↵
```

Figure 3. TEI markup corresponding to a single page of Luther's New Testament translation

3.2. Put Two-Column Regions into Separate FO Blocks

In a post to xsl-list quoted in [5], Eliot Kimber asked how to transform such a structure:

```
<fo:block span="all">
  <fo:block>
     <fo:block>
       <fo:block>
          <two-column-start/>
      </fo:block>
      ...
     <two-column-end/>
   </fo:block>
  <fo:block>...
  </fo:block>
</fo:block>
```

into something like this:

```
<fo:block span="all">
   <!-- Stuff before two-column start -->
</fo:block>
<fo:block span="none">
   <!-- Stuff up to <two-column-end/> marker -->
</fo:block>
```

```
<fo:block span="all">
  <!-- Stuff after <two-column-end> marker -->
</fo:block>
```

He assumed that there “must be a general pattern for solving this kind of transformation pattern” and he assumed that it involves grouping. The approach presented in this paper seems to be the solution he was looking for, and he used it successfully (with modification though) to tackle the issue.

The core template involves a nested `@group-starting-with/@group-ending-with` construct:

```
<xsl:template match="fo:block[empty(ancestor::fo:block)]" mode="#default">
  <xsl:variable name="block-root" as="element(fo:block)" select="."/>
  <xsl:for-each-group select="descendant::node()[empty(node())]"
    group-starting-with="two-column-start">
    <xsl:for-each-group select="current-group()"
      group-ending-with="two-column-end">
      <xsl:apply-templates select="$block-root" mode="split">
        <xsl:with-param name="restricted-to" as="node()*"
          select="current-group()/ancestor-or-self::node()" tunnel="yes"/>
        <xsl:with-param name="two-col-start" as="xs:boolean" tunnel="yes"
          select="exists(self::two-column-start)"/>
        <xsl:with-param name="pos" as="xs:integer" tunnel="yes"
                        select="position()"/>
      </xsl:apply-templates>
    </xsl:for-each-group>
  </xsl:for-each-group>
</xsl:template>
```

An additional parameter `$two-col-start` is passed that will inform the processing template whether the current group is a two-column area which necessitates a `span="none"` on the outermost `fo:block`, which is also the splitting root in this scenario.

The template that processes this outermost block in `split` mode looks as follows:

```
<xsl:template match="fo:block[empty(ancestor::fo:block)]" mode="split">
  <xsl:param name="restricted-to" tunnel="yes" as="node()*"/>
  <xsl:param name="two-col-start" tunnel="yes" as="xs:boolean"/>
  <xsl:copy>
    <xsl:apply-templates select="@*" mode="#current"/>
    <xsl:if test="$two-col-start">
      <xsl:attribute name="span" select="'none'"/>
    </xsl:if>
    <xsl:apply-templates mode="#current"/>
  </xsl:copy>
</xsl:template>
```

The actual FO problem was more complicated than sketched on the mailing list. Since some structures that may carry IDs will be duplicated within the same document, some ID fixup was necessary. Eliot mentions some more issues in his presentation at DITA OT day 2018 [6].

An animated demonstration of this processing using Saxon-JS is included in [4]. It will hopefully clarify or at least illustrate how this downward selection / grouping / upward projection / downward processing approach works.

3.3. Split at Line Breaks, Excluding Footnotes, List Items, etc.

For the next example we will look at content in the DocBook namespace. DocBook, like other XML vocabularies, features paragraphs that are indirectly nested in paragraphs – because they are contained in a list, a table, a blockquote, a footnote, or other things that are allowed to be contained in paragraphs. Now if we want to split a paragraph at line breaks, and if this paragraph also contains a list whose list items contain paragraphs with line breaks, only line breaks that belong to the paragraph itself should be considered, not line breaks that belong to the list items. Instead, the list item paragraphs should be split individually at their own line breaks.

You might argue that there is no such thing as a line break in DocBook. Well, there is. Sometimes people who want to encode line breaks work around this limitation by using something like `<phrase role="br"/>`, or in the DocBook Publishers derivative Hub XML [7], we allowed a `br` element proper in order to be able to better represent content from word processors and DTP programs.

Let's look at an example in which paragraphs with line breaks are considered as stanzas of a poem. DocBook Publishers knows a `linegroup` element that each paragraph is supposed to be transformed into. The split parts will be wrapped in `line` elements. Since line breaks are sometimes buried in other elements, such as `emphasis` or `phrase`, it is generally not feasible to do a plain grouping of all `para/node()`s.

As stated above, naïve upward projection, on the other hand, will split a paragraph at a line break in a contained list item or footnote, distributing the footnote between two `line`s that shouldn't have been separated in the first place. This is where "scoping" comes into play. A new scope is established in DocBook whenever there is an element that may be embedded in a `para` and that may itself contain `para`s.

We first introduce a variable that holds the local names of scope-establishing DocBook elements:

```
<xsl:variable name="dbk:scope-establishing-elements"
  as="xs:string+" select="('annotation',
                           'entry',
                           'blockquote',
```

```
                              'figure',
                              'footnote',
                              'indexterm',
                              'listitem',
                              'table',
                              'sidebar')"/>
```

(Yes, authors and typesetters manage to put line breaks into index terms, too…)

Then we introduce a function `dbk:same-scope()` that answers for a given node (text node, `emphasis` element, `br` element, …) and for a given paragraph whether both are in the same scope. To be precise: The function looks at the node's nearest scope-establishing ancestor element. Then it checks whether this scope-establishing element is among the paragraph's descendants. If it is, the node and the paragraph are in different scopes. This is the case if, for example, a footnote is in the way when looking upward from the node to the paragraph.

```
<xsl:function name="dbk:same-scope" as="xs:boolean">
  <xsl:param name="node" as="node()"/>
  <xsl:param name="para" as="element()"/>
  <xsl:sequence select="empty(
     $node/ancestor::*[local-name() = $dbk:scope-establishing-elements]
                     [1]
     intersect $para/descendant::*
   )"></xsl:sequence>
</xsl:function>
```

The transformation starts in mode `dbk:br-to-line` (a mode that reproduces most of the document identically). When it encounters a `para` or `simpara` (`$dbk:br-to-line-element-names`) element that contains `br` elements in the same scope, the following template will match:

```
<xsl:template match="*[local-name() = $dbk:br-to-line-element-names]
                     [
                      .//br[
                        dbk:same-scope(., current())
                       ]
                     ]" mode="dbk:br-to-line">
  <xsl:variable name="context" select="." as="element(*)" />
  <poetry><linegroup>
    <xsl:apply-templates select="@*" mode="#current"/>
    <xsl:for-each-group
      select="descendant::node()[
        local-name() = $dbk:scope-establishing-elements
        or (dbk:same-scope(., current()) and empty(node()))
      ]" group-starting-with="br">
      <xsl:apply-templates select="$context" mode="dbk:upward-project-br">
        <xsl:with-param name="restricted-to" tunnel="yes"
```

```
        select="current-group()/ancestor-or-self::node()"/>
    </xsl:apply-templates>
  </xsl:for-each-group>
  </linegroup></poetry>
</xsl:template>
```

This means that the scope-establishing elements will also be treated as leaf nodes. When such an element is encountered in mode `dbk:upward-project-br` (the mode that has the conditional identity template), it switches back to the original `dbk:br-to-line` mode:

```
<xsl:template mode="dbk:upward-project-br"
  match="*[local-name() = $dbk:scope-establishing-elements]">
  <xsl:param name="restricted-to" as="node()+" tunnel="yes" />
  <xsl:if test="exists(. intersect $restricted-to)">
    <xsl:apply-templates select="." mode="dbk:br-to-line" />
  </xsl:if>
</xsl:template>
```

This way, the paragraphs in footnotes or list items will be subjected to the same scope-aware splitting.

Given the following input:

```
<section xmlns="http://docbook.org/ns/docbook"
  version="5.1-variant le-tex_Hub-1.2">
  <title>DocBook Poetry</title>
  <para>Para without a line break.</para>
  <para>This is some sample text<br/>
    with line <emphasis>breaks,<br/>
  emphasis around line <phrase role="sc">breaks,<br/>
  and</phrase> a footnote</emphasis> with line breaks<footnote>
    <para>This footnote contains<br/>
    a line break.</para>
  </footnote>.<footnote>
    <para>This does not.</para>
  </footnote></para>
</section>
```

the stylesheet will create this output (modulo indentation):

```
<section xmlns="http://docbook.org/ns/docbook"
  version="5.1-variant le-tex_Hub-1.2">
  <title>DocBook Poetry</title>
  <para>Para without a line break.</para>
  <poetry>
    <linegroup>
      <line>This is some sample text</line>
      <line>with line <emphasis>breaks,</emphasis></line>
      <line><emphasis>emphasis around line
```

```
          <phrase role="sc">breaks,</phrase></emphasis></line>
        <line><emphasis><phrase role="sc">and</phrase> a
          footnote</emphasis> with line breaks<footnote>
            <poetry>
              <linegroup>
                <line>This footnote contains</line>
                <line>a line break.</line>
              </linegroup>
            </poetry>
          </footnote>.<footnote>
            <para>This does not.</para>
          </footnote></line>
      </linegroup>
    </poetry>
  </section>
```

The complete stylesheet, including this sample input, is again at [4].

4. Performance

The author's initial assumption was that the method scales roughly linearly with the number of nodes and with the number of splitting points. This would result in a roughly quadratic dependence on input length. However, measurements do not corroborate this. The method seems to scale roughly linearly with input length, but also roughly linearly with chunk size. The latter means that the *fewer* splitting points there are for a given document size, the longer will the splitting take.

Figure 4 shows the execution times for the TEI data from Section 3.1. The original input was transformed with a variation of the chunker that accepts the target page count as a parameter and then splits before the corresponding `pb` element. It will write only the first chunk into a file.

"Constant chunk length" is not entirely correct. Of course there are variations in string lengths and node count between the pages. The first 8 pages contain almost no content which explains the slight outlier for 10 chunks (the leftmost data point).

Note

All measurements have been carried out with Saxon EE 9.9.1.1 with `-opt:9` optimization on an AMD PRO A12-9800B R7 CPU at 2.7 GHz on a 64-bit Windows 10 operating system and a 64-bit Oracle Java 1.8.0_151 JVM. For transformations that take less than 10 seconds to complete, average times for 6 runs have been calculated using Saxon's `-repeat` option. For longer-running transformations (up to a minute), at least 3 runs have been averaged. For processes that ran longer than a minute, at least two runs have been averaged.

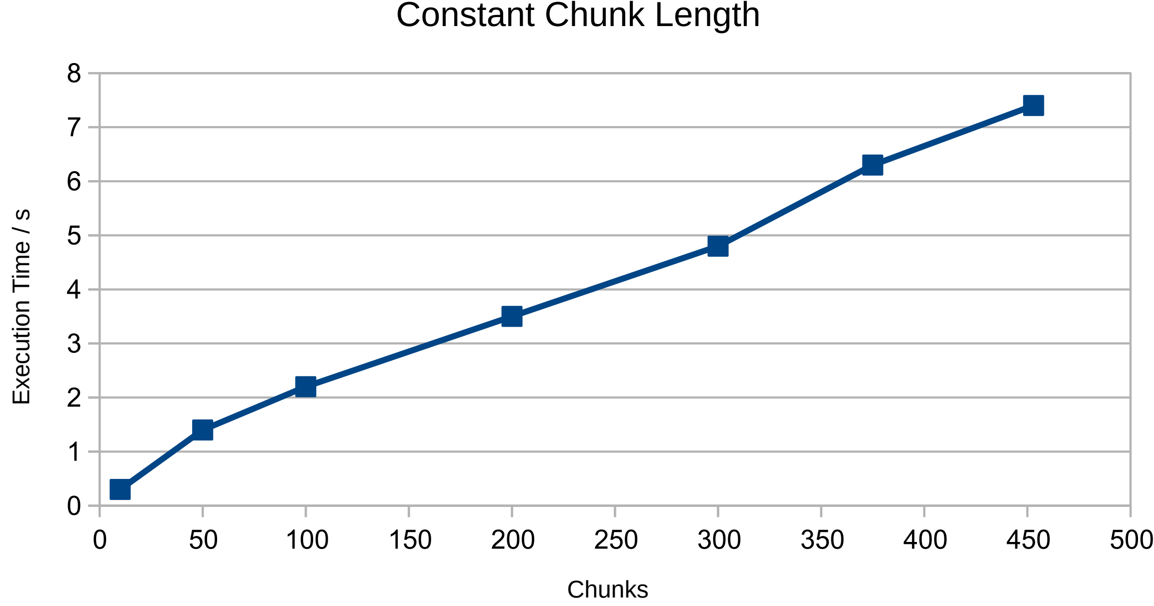


Figure 4. Execution times for roughly constant chunk sizes

When creating the shortened documents with the modified upward projection stylesheet, the author noticed that execution time would grow disproportionally with the target page count, that is, with chunk size. Creating the 375-page document took nearly 12 minutes.

Therefore the author created another test set series by repeatedly removing every other `pb` element while keeping the rest of the document unchanged, until there were only 4 page breaks left, spread evenly over the document. Figure 5 displays the measurements.

If execution times are displayed in relation to average chunk size (in terms of the number of leaf nodes per chunk), the graph will enter an approximately linear regime for large chunks, as seen in Figure 6.

Profiling the conversion runs for different chunk sizes revealed that this time can be attributed almost exclusively to invocations of the conditional identity template that receives the tunneled parameter `$restricted-to`. The total number of invocations of this template did not change dramatically, from about 322,000 for 453 chunks down to roughly 126,000 for 58 chunks and 105,000 invocations for 5 chunks. But the average time for this template increased from 0.039 ms (453 chunks) to 3.174 ms (note that the times measured during profiling are larger than normal, so the multiplied/added times do not exactly match the execution times given above).

The measurements have been made with an important optimization already in place. The tunneled parameter `$restricted-to` has been changed from

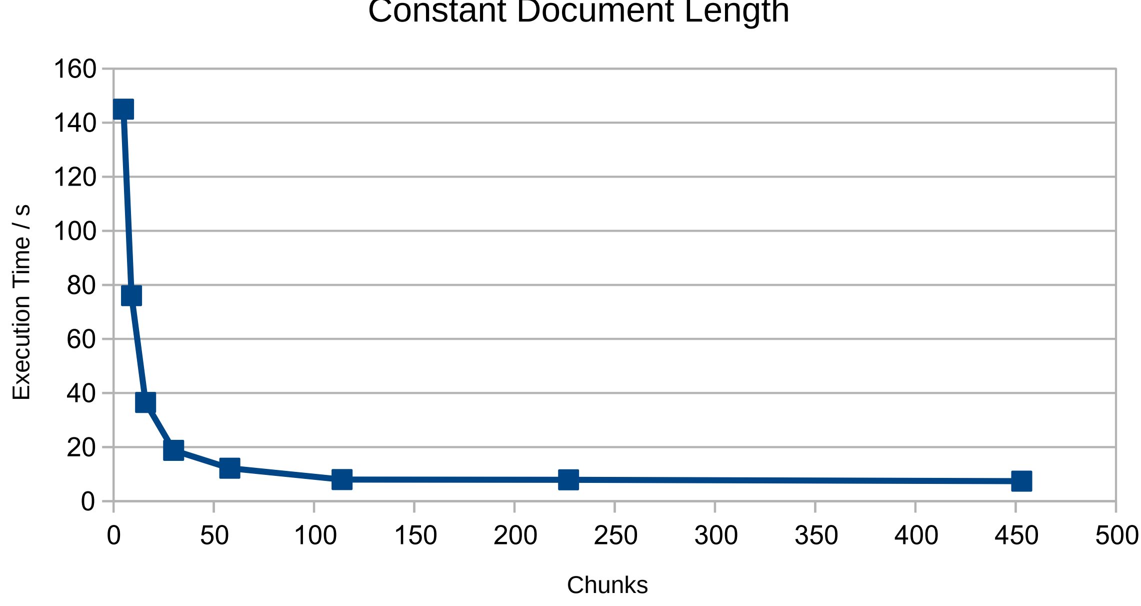


Figure 5. Execution times for constant document length

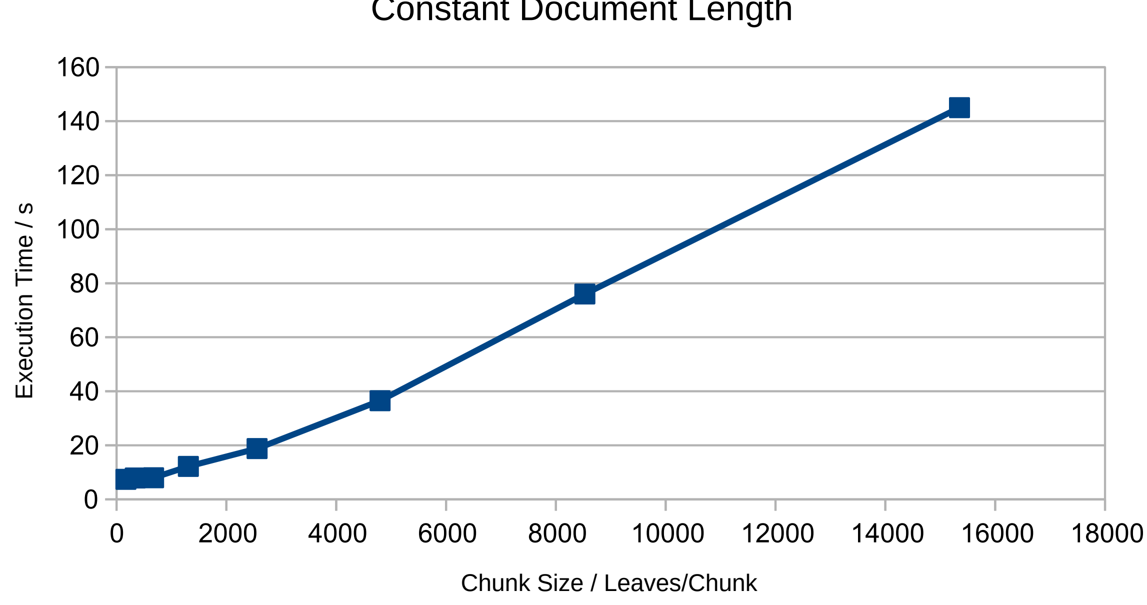


Figure 6. Execution times for constant document length (in relation to average chunk size)

sequence type `node()+` to `xs:string+` with the string values computed by `generate-id()`. The conditional identity template has been changed accordingly:

```
<xsl:template match="node()" mode="split" priority="1">
  <xsl:param name="restricted-to" as="xs:string+" tunnel="yes"/>
  <xsl:if test="generate-id() = $restricted-to">
    <xsl:next-match/>
  </xsl:if>
</xsl:template>
```

At least for Saxon 9.9, `generate-id()` lookups in a sequence of tens of thousands of strings seems to perform about 20 times better than the previously used `exists(. intersect $restricted-to)` approach.

The lookup time seems to grow linearly with sequence size, which explains the linear growth for large chunk sizes. For smaller chunk sizes, the fact that the conditional identity template had to be called more often (because some nodes are visited multiple times: when creating each chunk) explains that the left part of the graph shows larger execution times than an extrapolation of the linear part would suggest. So the number of splitting points does have an effect, but it is a less-than-linear effect.

It must be said that for larger depths at which the `pb` elements are buried, slightly worse performance measurements may be expected (because more elements will be visited for each chunk). But the XPath distribution of the `pb`s in the TEI sample document is probably quite typical, therefore the measurements can be regarded as representative.

In addition, if the document were more fragmented in terms of the number of leaf nodes, performance would deteriorate more dramatically, but again not worse than proportional to the number of leaves per chunk.

5. Applicability of XSLT 3.0 Features: Dynamic XPath Evaluation, Streaming

5.1. Streaming

There has been a brief discussion in 2014 about how this method is problematic with streaming [8]. Although the processor is now much more mature, nothing has changed with respect to the intrinsic lack of streamability of this method. Quoting Michael Kay: "… the real problem is that the logic is going down to descendants, then up to their ancestors, and then down again, and that's intrinsically not processing nodes in document order, which is a precondition for streaming."

And even if it were streamable, it would probably be ill-suited to process large documents with a large number of leaf nodes because the performance, as discussed in the previous section, deteriorates proportionally to the number of leaf nodes.

Nevertheless, one may still use the solution in non-streaming modes within streaming scenarios. But chunking large documents must be left to other methods.

5.2. Dynamic XPath Evaluation

As a proof of concept, the author created a configurable chunker for `group-starting-with` splittings as in the TEI `pb` scenario. The leaf selection XPath expression, the splitting point matching pattern and another parameter (how to name the resulting chunks) can be given as an XPath expression that will be dynamically evaluated. In addition, a parameter can trigger whether the splitting element should be kept or not. The complete XSLT package looks like this:

```
<xsl:package
    name="http://www.le-tex.de/XSLT/split"
    package-version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:split="http://www.le-tex.de/XSLT/split"
    exclude-result-prefixes="xs split"
    version="3.0">

  <xsl:mode name="split:split" visibility="private"/>
  <xsl:mode name="split:split-entrypoint" visibility="final"/>

  <xsl:template match="*" mode="split:split-entrypoint">
    <xsl:param name="leaves-exp" as="xs:string"
      select="'outermost(descendant::node()[not(has-children(.))])'"/>
    <xsl:param name="group-start-exp" as="xs:string"/>
    <xsl:param name="chunk-name-exp" as="xs:string"/>
    <xsl:param name="keep-splitting-node" as="xs:boolean" select="true()"/>
    <xsl:document>
      <split:chunks>
        <xsl:variable name="context" select="." as="element()"/>
        <xsl:for-each-group select="split:eval-xpath(., $leaves-exp)"
          group-starting-with="node()[split:node-matches-xpath(.,
                                $group-start-exp)]">
          <split:chunk>
            <xsl:attribute name="n" select="position()"/>
            <xsl:if test="$chunk-name-exp">
              <xsl:attribute name="name">
                <xsl:evaluate xpath="$chunk-name-exp" context-item="."/>
              </xsl:attribute>
            </xsl:if>
            <xsl:apply-templates select="$context" mode="split:split">
              <xsl:with-param name="restricted-to" tunnel="yes"
```

```
                select="for $n in (
                          current-group()/ancestor-or-self::node()
                          except .[not($keep-splitting-node)]
                        )
                        return generate-id($n)"/>
            </xsl:apply-templates>
          </split:chunk>
        </xsl:for-each-group>
      </split:chunks>
    </xsl:document>
  </xsl:template>

  <xsl:function name="split:node-matches-xpath" as="xs:boolean"
    visibility="private">
    <xsl:param name="_node" as="node()"/>
    <xsl:param name="xpath" as="xs:string"/>
    <xsl:sequence select="exists(split:eval-xpath($_node, $xpath))"/>
  </xsl:function>

  <xsl:function name="split:eval-xpath" as="item()*" visibility="private">
    <xsl:param name="context" as="node()"/>
    <xsl:param name="xpath" as="xs:string"/>
    <xsl:evaluate xpath="$xpath" context-item="$context" as="node()*"/>
  </xsl:function>

  <xsl:template match="@* | node()" mode="split:split" priority="-1">
    <xsl:copy>
      <xsl:apply-templates select="@*, node()" mode="#current"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="node()" mode="split:split">
    <xsl:param name="restricted-to" as="xs:string+" tunnel="yes"/>
    <xsl:if test="generate-id() = $restricted-to">
      <xsl:next-match/>
    </xsl:if>
  </xsl:template>
</xsl:package>
```

XSLT packages are ideally suited for providing this as re-usable code. Without the visibility restrictions that packages offer, XSLT authors who import this code may always overwrite templates. This should be avoided because an adaptation that is good for one use case might not be suitable for another, thereby jeopardizing reusability. Therefore, dynamic evaluation together with visibility constraints are perfect for providing a generic splitter.

Applying it to the TEI pb use case requires a modification of the front-end stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tei="http://www.tei-c.org/ns/1.0"
  xmlns:split="http://www.le-tex.de/XSLT/split"
  xpath-default-namespace="http://www.tei-c.org/ns/1.0"
  exclude-result-prefixes="xs split"
  version="3.0">

  <!-- invoke it like this:
    saxon-EE-9.9.1.1 -it:test -lib:lib/split.xsl
      -s:examples/pb/luther_septembertestament_1522.TEI-P5.xml
      -xsl:examples/pb/split-at-pb_1_generate-id_evaluate.xsl -->

  <xsl:use-package name="http://www.le-tex.de/XSLT/split"
    package-version="1.0"/>

  <xsl:template match="/">
    <xsl:variable name="chunks" as="document-
node(element(split:chunks))">
      <xsl:apply-templates select="TEI/text" mode="split:split-
entrypoint">
        <xsl:with-param name="group-start-exp" as="xs:string"
          select="'self::Q{http://www.tei-c.org/ns/1.0}pb[@facs]'"/>
        <xsl:with-param name="chunk-name-exp" as="xs:string"
          select="'substring(@facs, 3)'"/>
        <xsl:with-param name="keep-splitting-node" select="true()"/>
      </xsl:apply-templates>
    </xsl:variable>
    <xsl:sequence select="$chunks"/>
    <xsl:variable name="tei-top" as="element(TEI)" select="/TEI"/>
    <xsl:for-each
      select="$chunks/split:chunks/split:chunk[text()[normalize-space()]
| *]">
      <xsl:variable name="chunk" select="."/>
      <xsl:result-document href="out/dyn{@name}.xml">
        <xsl:copy select="$tei-top" copy-namespaces="false">
          <xsl:copy-of select="@*, teiHeader, $chunk/node()"
            copy-namespaces="false"/>
        </xsl:copy>
      </xsl:result-document>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The execution times are roughly 1.5 to 2.1 times as long as in the non-generic case which makes the generic solution attractive especially for ad-hoc processing. For large-scale productions, it will still be better to use bespoke solutions that are targeted at specific vocabularies and splitting problems.

The generic solution will be extended in order to cover start-end delimiters as in the FO scenario, and also by providing a dynamic evaluation facility for selecting/matching scope-establishing elements.

6. Summary

An elegant and versatile XML splitting solution has been presented. It is applicable to a wide range of splitting problems, as proven by its usage in le-tex publishing services' XSLT conversions since 2010. While preparing this paper, the author was able to identify and rectify performance limitations. Other limitations, in particular when processing large documents or chunks with many (~10^5) leaf nodes, could not be surmounted though.

Bibliography

[1] tokenized-to-tree: An XProc/XSLT Library For Patching Back Tokenization/ Analysis Results Into Marked-up Text. In: Proceedings of XML Prague 2018, http://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf#page=241.

[2] Example involving page breaks: http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201407/msg00004.html.

[3] Martin Luther (translator): *Das Newe Testament Deutzsch. [Septembertestament.]* Wittenberg, 1522. In: Deutsches Textarchiv `http://www.deutschestextarchiv.de/book/show/luther_septembertestament_1522`

[4] Source code repository for this paper: `https://subversion.le-tex.de/common/presentations/2019-02-09_xmlprague_xslt-upward-projection`

[5] Example involving XSL-FO blocks: https://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201804/msg00044.html.

[6] A video of Eliot's presentation is linked at `https://www.oxygenxml.com/events/2018/dita-ot_day.html#twisted_xslt_tricks`

[7] Hub XML: `https://github.com/le-tex/Hub`

[8] Discussion about the suitability for streaming in 2014: https://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201407/msg00036.html.

Sonar XSL

A Schematron-based SonarQube plugin for XSL code quality measurement.

Jim Etevenard
OXiane
<jetevenard@oxiane.com>

Abstract

While code quality measurement tends to be wildly used in software engineering, no suitable tool were available for XSL Language. This paper presents the Sonar XSL project, a plugin that performs XSLT quality measurement in SonarQube, with Schematron under the hood.

Keywords: SonarQube, Schematron, XSL, code quality

1. The Necessity of Code Quality Measurement

XSL language is reaching a maturity level that allows using it for complex and big projects. In that context, developers must keep in mind that their code should conform to a certain quality level.

The Continuous Integration (CI) and Continuous Deployment (CD) processes tends to become the standard in code industry. In such a context, continuous quality exigence and measurement become essentials.

Code quality should be a constant concern of any developer. In this purpose, it should be constantly measured, and the result of that measurement should take part into any developer team's workflow: as well as the unit tests, the project's conformance to a shared quality standard should be part of the *definition of done*.

1.1. The Axes of code Quality

First, we must precise that the notion of "Quality" is intrinsically subjective. By "Quality of code", we mean conforming to a certain number of criteria that will mainly tend to reduce the risks of failure of an application, or improve it's maintainability and so it's sustainability in the long term.

These criteria must be team-shared and can - and must - be discussed. As well as craftsmen, developers must be involved in the process of defining what is called "Good work".

A lot of work and studies have been done around code quality by computer scientists around the world, and a consensus has emerged over seven main Code Quality axes:

1.1.1. Tests and testability

Any code unit module (which, depending on the language, we may call *function*, *method*, *template*...) should be covered by unit and/or integration tests. The important thing is not to just call 100% of these *modules*, but to ensure that all the instructions are effectively executed, and that as much failure cases as possible are tested.

This purpose implies to design the code and distribute the complexity in a way that optimises the *testability* of the code. Any low-level module should have a clearly-defined behavior for a given input. Any environment-dependant operations like data access, TCP requests, file management should be as much separated as possible to preserve testability.

Some coding management frameworks like the Test-Driven Developpement suggest to first develop the test, defining the input and the expected output, to then run the test and to see it fail, and to finally develop the simplest way possible to make it succeed. This process puts the tests in the heart of development and strongly increase the reliability of an application.

Unit tests can also be seen as an in-code documentation, since they describe the expected behavior of a code module.

1.1.2. Duplicated code

Code duplication is one the most frequent mistake in computer programming, trough it is probably one of the worst.

When time is missing, there can be a temptation to quickly duplicate code parts instead of refactoring to distribute and share the complexity.

Duplicating code blocks - or, in a much more subtle and harder to detect form, having several code parts that do the same thing - is a serious problem for maintainability: if an evolution is needed in the duplicated process, the changes will have to be duplicated. That is, of course, a loss of time, and having duplications makes it difficult to localise the code parts that must evolve.

1.1.3. Potential bugs

Since every programming language has its pitfalls, some coding patterns are known for frequently leading to put bugs in an application, or expose it to some known vulnerabilities.

These risks may come, for example, from a developper's lack of attention and/or knowledge of language specificities: exact role of some operators, type-conversion behavior, etc. Some frameworks or libraries may bring some vulnerabilities when incorrectly used.

1.1.4. Complex code

Code is not only intended for computers, but also for humans : it should be clear, and esay to read and unnderstand for further debuging and/or evolving operations.

Its cyclomatic complexity should be reduced as much as possible to keep it easy to understand and testable. Modules that contain many nested conditional and/or loop structures could favorably be split into lower-granularity modules that would be simpler.

1.1.5. Architecture and Design

The lack of a clearly-defined, understandable and easy-to-conform architecture is likely to lead to what is commonly called "Spaghetti code": Something that won't be structured, and thus hard to maintain, and where errors or bugs will be hard to detect and fix.

1.1.6. Documentation and Comments

Commenting / documenting code makes it easy to understand, use or extend it. Especially if a given code part is the end point of an API, intended to be called from another application or module.

In-code documentation is now a standard in the imperative languages world: Javadoc, in the Java-world, has popularised the concept of improving the comments with markup tags that allows automated easy-to-consult document generation. This approach has been reworked in many javadoc-like framework like Doxygen, JSDoc, pydoc, etc. in various programming languages.

In the XML world, booth XSD and RelaxNg schema languages provide standard documentation patterns. More recently, some documentation schemes have been released for XSL.

2. The Existing Standard : SonarQube

SonarQube is an open-source software that statically analyses source code and pull up various metrics regarding code duplication, unit test coverage, documentation level and coding rules. By statically we mean performing an analysis of the source code without executing it.

It comes from the Java world, and it itself coded in Java.

SonarQube alone is not able to analyse anything : it's architecture relies on plugins that are responsible for analysing the code. Out of the box, a dozen plugins are embedded in SonarQube, allowing it to analyse projects in the main programming languages: Java, PHP, Js, C#, C, Css, Etc.

2.1. The main concept : Rules

These plugins define some *Rules* for the language they focus on. These *Rules* are one of the main concepts of SonarQube.

A *Rule* consist in a best-practice pattern that should be respected in the source code. Something important to understand about SonarQube and his community is that a *Rule* is not something intended to be strongly imperative. A *Rule* is a community-consensual proposal of what a quality involved developer should, or should not do. These *Rules* are meant to be discussed, and SonarQube allows the instance manager to disable some rules.

Rules are documented with a description, for which the SonarQube community has designed a writing pattern[1]. This documentation should explain the reasons why the *Rule* should be followed, and provide and non-compliant and a compliant example.

They had a strong reflexion about the wording to use in the Rule statement and description: Negative and/or imperative form should be avoided, the main idea is that the *Rules* are best-practice recommendations, and not commands falling from Mount Olympus.

The *Rules* hold some meta-information that are useful for handling the analysis result.

- Their type
  - **Code smell** – Code best-practice and design
  - **Bug** – Potential bug
  - **Vulnerability** – Potential Vulnerability
- Their (default) severity
  - Blocker
  - Critical
  - Major
  - Minor
  - Info
- Some subject-related keywords

The *Rules* come with a default severity, which can be overridden through the settings.

2.2. Dealing with Issues

The detection, in the analysed code, of some code parts that breaks a *Rule* creates an *Issue*.

The result of a code analysis is a list of *Issues*, which are linked to their source *Rule*, and contextualized in source code.

[1] https://docs.sonarqube.org/display/DEV/Coding+Rule+Guidelines

Ideally, an analysis should be run automatically each time changes are committed to the SCV. SonarQube makes a distinction between *Issues* that are raised from *new code* (i.e. code added with last commit) and those which were already there during a previous analysis of the project. This distinction is important since we rarely have the luck to begin a project from scratch, and the approach to improve the code quality of an application must consider the *real-life reality* of the existing legacy code.

The SonarQube's API allows to connect this *Issue* system with project management tools like Jira, Mantis, Trello and so on, to integrate the correction of the *Issues* into the team workflow.

The *Issues* screen itself, that allows filtering through *Issues* and provide an assignment system, is sufficiently powerful to be itself considered as a project management system.

2.3. The verdict : Quality Gates

Afted each analysis of the project, the *Issues* metrics are compiled through a *Quality gate*. The *Quality gate* consists in acceptance critereas defined for the project.

Examples :

- At least 80% of the new code should be covered by unit tests
- There should not be *Major* or *Blocker Issues* in the project
- There should be less *Issues* than during the previous analysis
- Etc.

2.4. Whats SonarQube Brings

SonarQube is a tool intended not only for developers, but also for non-technical managers.

The concepts of "code quality", "reliability", "maintainability", "technical debt", etc. are very abstract for non-technical people.

By raising metrics, SonarQube helps project managers to have a visibility on these concepts and the needed work to enforce the quality of their applications.

In the exigent context of agile development and continuous delivery, SonarQube provides a good way to ensure the quality of what is delivered.

2.5. The lack of an XSL-Specific SonarQube plugin

While it is nowadays currently used for enterprise applications written in the above-mentioned languages, no SonarQube plugins that add specific support for XSL were available so far. (Saying that, I'm excluding the Sonar-XML-Plugin that performs the strict-minimal well-formed / validity tests)

3. The XSLT-Quality Schematron

Some XML developers addressed the question of XSL Quality :

- Mukul Gandhi[2] released in 2009 his XSL Quality XSLT[3], a tool written in XSLT 2.0 that checks the conformance of an XSL Stylesheet to a set of rules that he defined.
- Matthieu Ricaud-Dussarget published last year an ISO-Schematron to check the code-quality of an XSLT[4].
  - He re-wrote Mukul Gandhi's rules as Schematron asserts / reports
  - He added some rules from his own work.

These rulesets are intended to help developers to avoid the common pitfalls of XSL.

3.1. Examples of Rules from XSLT-Quality

- Variables should be typed
- The use of wildcard namespaces `*:` in xpath statements should be avoided
- Using `xsl:for-each` on nodes may be a clue of procedural programming ; You should maybe consider using `xsl:apply-templates` instead
- boolean `true()` or `false()` value should be used instead of litteral strings 'true' and 'false'
- Costly double-slash operator should be used with caution
- Global variables, global parameters and mode names should have a namespace to improve the portability of your stylesheet
- etc.

4. The Sonar-XSL-Plugin Prototype

Some coding rules exist for XSL, written as a Schematron : why not build a Sonar Plugin that validates the XSL code using that Schematron ?

I've created an open-source SonarQube plugin for XSL, that performs some code-style checks, using the XSLT-Quality[5] Schematron project mentioned above.

The implementation is based on Schematron's official "Skeleton" XSLT implementation[6], and Saxon[7] as XSLT processor.

The *asserts* and *reports* from the Schematron documents are transcribed into SonarQube *Rules* and registered into Sonar through its API.

---

[2] https://www.xml.com/authors/mukul-gandhi/
[3] http://gandhimukul.tripod.com/xslt/xslquality.html
[4] https://github.com/mricaud/xslt-quality
[5] https://github.com/mricaud/xslt-quality
[6] https://github.com/Schematron/schematron/tree/master/trunk/schematron/code
[7] http://saxon.sourceforge.net/

When a code analysis is launched, each XSL file is validated against the Schematrons : If a Schematron *assert* fails, or if a *report* is triggered, an *Issue* is created in the SonarQube system. The localisation of that *Issue* in the file, a contextual message and any useful additional are saved with that *Issue*.

4.1. Architecture of the Plugin

The plugin is built using Maven. It consists in several modules :

4.1.1. The Schematron-Sonar Module

The main reactor, it is responsible for reading and running the Schematron grammars, and establishing the link with SonarQube API.

This module can be re-used to build a SonarQube plugin for any XML-based language.

4.1.2. Some Schematron packages

These artifacts contain the Schematron used for code quality checks.

- The Sonar XSL Plugin embeds the XSLT-Quality[8] mentioned above as it's main rule repository.
- A Maven plugin is used to build the *Schematron packages*, organizing the result packages (which consists in a *Jar* archive) the way expected by the *Schematron-Sonar Module*.
- At a project / enterprise level, one can re-built his own flavor of the *Sonar XSL Plugin*, adding his own *Schematron packages*.

5. Evolutions and Perspectives

The *Sonar XSL Plugin* will has been released a few weeks ago. Some code improvements are still needed.

It will soon be experimentally deployed in the company where I work - a publishing house - where we strongly rely on XSL code to exploit our content.

An important feature would be to measure the unit tests coverage. For that purpose, a further evolution of the *Sonar XSL Plugin* must be able to fetch informations from the test reports generated by the XSpec Maven Plugin[9].

Since coding rules are intrinsically subjective : a work on the rules themselves will remain for the XSL community. The rules need to be thank about and discussed, for a better quality in XSLT-based project.

[8] https://github.com/mricaud/xslt-quality
[9] https://github.com/xspec/xspec-maven-plugin-1

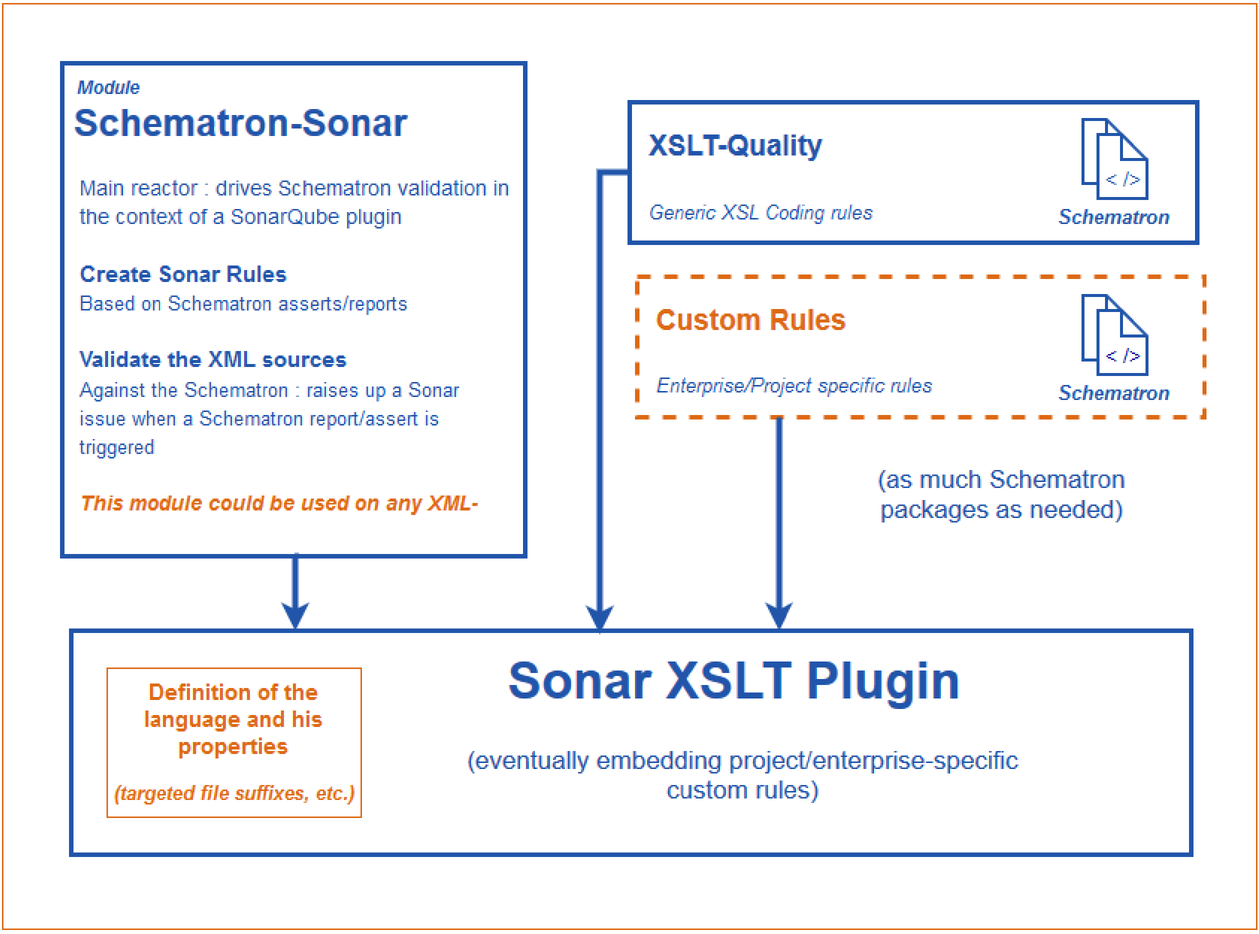


Figure 1. Architecture of the Sonar XSL Plugin

Copy-fitting for Fun and Profit

Tony Graham
Antenna House, Inc.
<tony@antennahouse.com>

Abstract

Copy-fitting is the fitting of words into the space available for them or, sometimes, adjusting the space available to fit the words. Copy-fitting is included in "Extensible Stylesheet Language (XSL) Requirements Version 2.0" and is a common feature of making real-world documents. This talk describes an ongoing internal project for automatically finding and fixing problems that can be fixed by copy fitting in XSL-FO.

1. Introduction

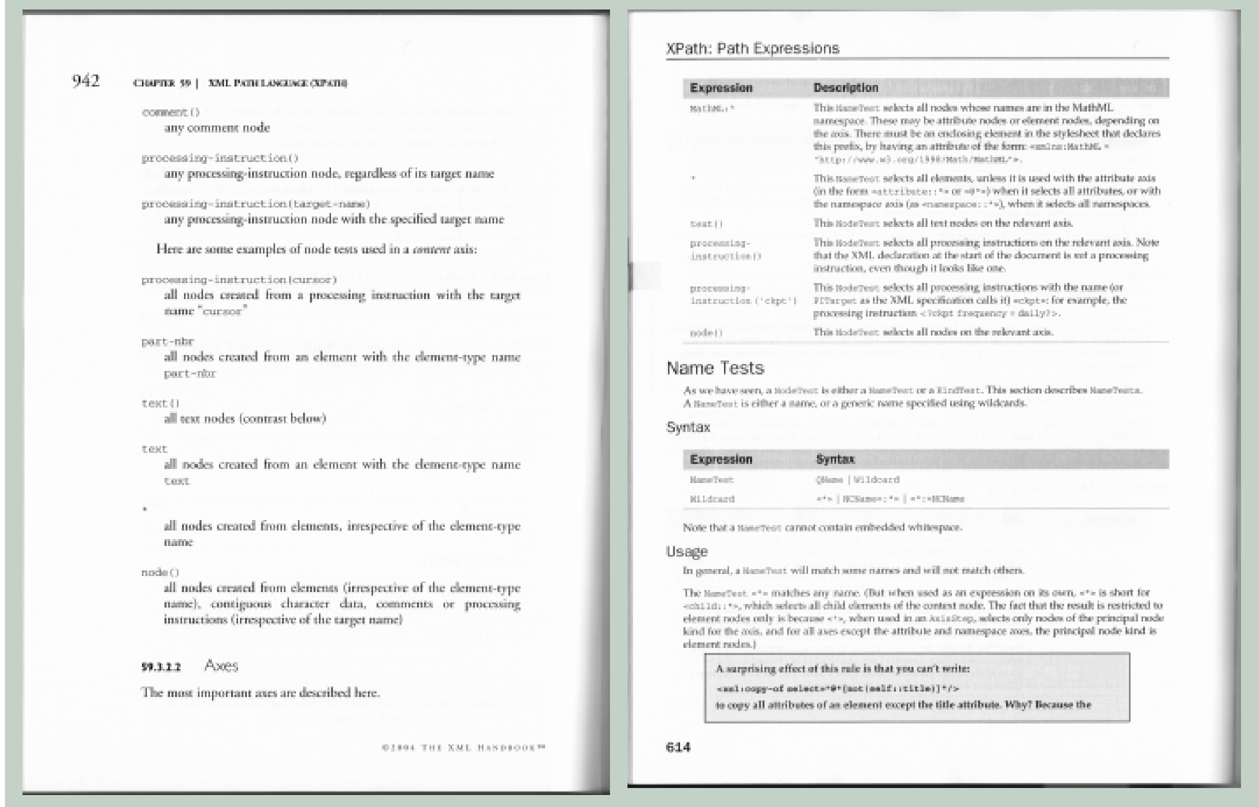
Copy-fitting has two meanings: it is both the "process of estimating the amount of space typewritten copy will occupy when converted into type" [7] and the process of adjusting the formatted text to fit the available space. Since automated formatting, such as with an XSL-FO formatter, is now so common, a lot of the manual processes for estimating the amount of space are superfluous.

1.1. Copy-fitting as estimating

There are multiple, and sometimes conflicting, aspects to the relationship between copy-fitting and profit. For the commercial publisher, there is tension between more pages costing more money and more pages providing a better reading experience or even a better "shelf appeal", for want of a better term. We tell ourselves "do not judge a book by its cover", but we do still sometimes judge a book by the width of its spine when we look at it on the shelf in the bookstore. Copy-fitting, in this first sense, is part of the process of making the text fill the number of pages (the 'extent') that has been decided in advance by the publisher. This may mean increasing the font-size, leading, other spaces, and the margins to fill more pages, or it may mean reducing them so that more text fits on a page.

The six steps to copy-fitting (as estimating) from "How to Spec Type" [7] are:

1. *Count manuscript characters*
2. *Select the typeface and type size you want.*
3. *Cast off (determine the number of set characters per line)*
4. *Divide set characters per line into total manuscript character count. Result is number of lines of set type.*



942 CHAPTER 59 | XML PATH LANGUAGE (XPATH)

comment()
any comment node

processing-instruction()
any processing-instruction node, regardless of its target name

processing-instruction(target-name)
any processing-instruction node with the specified target name

Here are some examples of node tests used in a *content* axis:

processing-instruction(cursor)
all nodes created from a processing instruction with the target name "cursor"

part-nbr
all nodes created from an element with the element-type name part-nbr

text()
all text nodes (contrast below)

text
all nodes created from an element with the element-type name text

*
all nodes created from elements, irrespective of the element-type name

node()
all nodes created from elements (irrespective of the element-type name), contiguous character data, comments or processing instructions (irrespective of the target name)

59.3.2.2 Axes

The most important axes are described here.

©2004 THE XML HANDBOOK™

XPath: Path Expressions

Expression	Description
MathML:*	This NameTest selects all nodes whose names are in the MathML namespace. These may be attribute nodes or element nodes, depending on the axis. There must be an enclosing element in the stylesheet that declares this prefix, by having an attribute of the form: «xmlns:MathML = "http://www.w3.org/1998/Math/MathML"».
*	This NameTest selects all elements, unless it is used with the attribute axis (in the form «attribute::*» or «@*») when it selects all attributes, or with the namespace axis (as «namespace::*»), when it selects all namespaces.
text()	This NodeTest selects all text nodes on the relevant axis.
processing-instruction()	This NodeTest selects all processing instructions on the relevant axis. Note that the XML declaration at the start of the document is not a processing instruction, even though it looks like one.
processing-instruction('ckpt')	This NodeTest selects all processing instructions with the name (or PITarget as the XML specification calls it) «ckpt»: for example, the processing instruction <?ckpt frequency = daily?>.
node()	This NodeTest selects all nodes on the relevant axis.

Name Tests

As we have seen, a NodeTest is either a NameTest or a KindTest. This section describes NameTests. A NameTest is either a name, or a generic name specified using wildcards.

Syntax

Expression	Syntax
NameTest	QName | Wildcard
Wildcard	«*» | NCName«:*» | «*:»NCName

Note that a NameTest cannot contain embedded whitespace.

Usage

In general, a NameTest will match some names and will not match others.

The NameTest «*» matches any name. (But when used as an expression on its own, «*» is short for «child::*», which selects all child elements of the context node. The fact that the result is restricted to element nodes only is because «*», when used in an AxisStep, selects only nodes of the principal node kind for the axis, and for all axes except the attribute and namespace axes, the principal node kind is element nodes.)

A surprising effect of this rule is that you can't write:

<xsl:copy-of select="@*[not(self::title)]"/>

to copy all attributes of an element except the title attribute. Why? Because the

614

Figure 1. Different approaches to filling pages [4][5]

5. *Add leading to taste.*

6. *If too short: add leading or increase type size or decrease line length.*
   If too long: remove leading or decrease type size or increase line length.

However:

> *Most important of all, decisions should be made with the ultimate aim of benefiting the reader.*
>
> *—"Book Typography: A Designer's Manual", Mitchell & Wightman*

1.2. Copy-fitting as adjustment

'Copy-fitting' as adjustment is now the more common use for the term.

2. Copy-fitting for fun

It's an interesting problem to solve.

3. Copy-fitting for profit

3.1. Books

The total number of pages in a book is generally a multiple of 16 or 32, and any variation can involve additional cost:

> *It must be remembered that books are normally printed and bound in sheets of 16, 32, 64 (or multiples upwards) pages, and this exact figure for any job must be known by the designer at the point of designing: it will be the designer's job to see that the book makes the required number of pages exactly. If a book is being printed and bound in sheets of 32 pages (16 on each side of the sheet), it is generally possible to add one leaf of 2 pages by 'tipping in', or 4, 8, or any other multiple of 4 pages extra by additional binding operation, but the former will mean hand-work, and even the latter will involve disproportionately higher cost.*
>
> *—"The Thames and Hudson Manual of Typography", Ruari McLean, 1980*

The role here for copy-fitting (in the second sense) is to ensure that the overall page count is at or close to a multiple of the number of pages in a signature.

Even Print-On-Demand (POD) printing has similar constraints. For example, the Blurb POD service requires that books in "trade" formats – e.g. 6×9 inches – are a multiple of six pages [1]. In a simplistic example, suppose that the document with the default styles applied formats as 15 pages:

Figure 2. Document with default styles and six-page signatures

The (self-)publisher has the choice of three alternatives:

- Leave the layout unchanged. However, the publisher is paying for the blank pages, and the number of blank pages may be more than the house style allows.
- Copy-fit to reduce the page count to the next lower multiple of the signature size. This, obviously, is cheaper than paying for blank pages, but “the public still has a tendency to judge the value of a book by its thickness” [8].

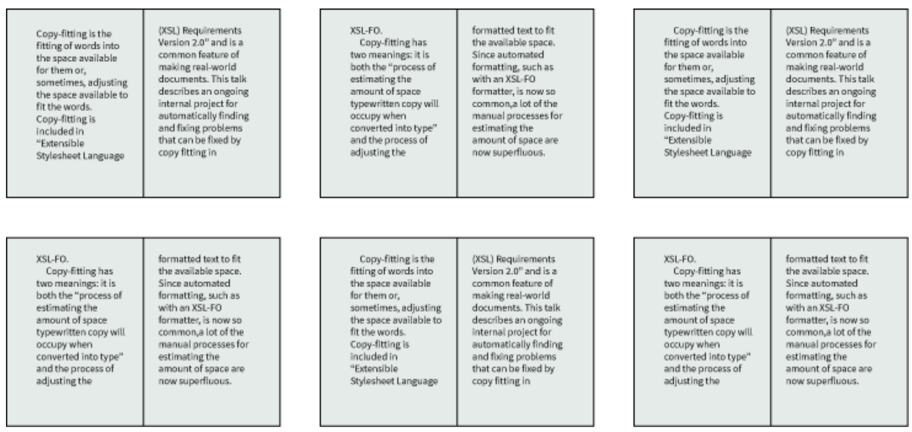


Figure 3. After copy-fitting to reduce page count

- Copy-fit to increase the page count to better fill a multiple of the signature size. This, just as obviously, costs more than reducing the page count, but it has the potential for helping sales.

Figure 4. After copy-fitting to increase page count

3.2. Manuals and other documentation

A manufacturer who provides printed documentation along with their product faces a different set of trade-offs. The format for the documentation may be constrained by the size of the product and its packaging, regulatory requirements, or the house style for documentation of a particular type. A manufacturer may need to print thousands or even millions[1] of documents. Users expect clear documentation yet may be unwilling to pay extra when the documentation is improved, yet unclear documentation can lead to increased support costs or, in some cases, to fatalities.

Suppose, for example, that the text for a document has been approved by the subject matter experts – and, in some cases, also by the company's lawyers and possibly the government regulator – and has to be printed on a single standard-size page, or possibly a single side of a standard-size page, yet there is too much text for the space that is available. To let the editorial staff rewrite the text so that it fits the available space is definitely not an option, so it becomes necessary to apply copy-fitting to adjust the formatting until the text does fit. Figure 5 shows an information sheet that was included with the Canon MP830 when sold in EMEA. The same information in 24 languages is printed on a single-sided sheet of paper. Most users would look at the information once, at most. If the information had been allowed to extend to the back of the sheet, it would have considerably increased the cost of providing the information for no real benefit. Some form of copy-fitting was probably used to make sure that the information could fit on one side of the sheet.

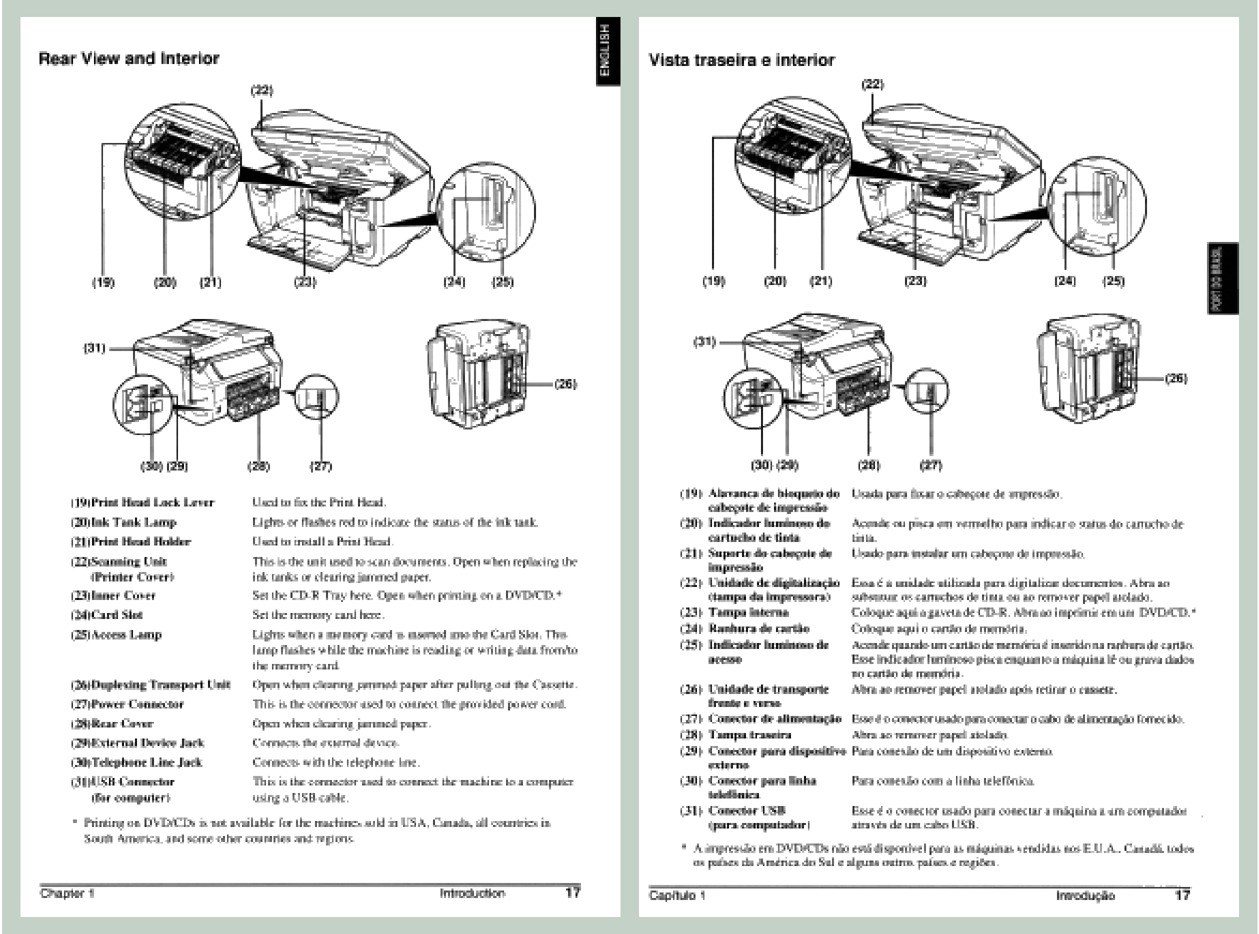
Multilingual text has other complications. Figure 6 shows two corresponding pages from the English and Brazilian Portuguese editions of the Canon MP830 "Quick Start Guide". The same information is presented on both pages. However, translations of English are typically longer than the corresponding English, and this is no exception. The Brazilian Portuguese page has more lines of text on it and, as Figure 7 shows, the font-size and leading has also been reduced in the Brazilian Portuguese page. A copy-fitting process could have been used to adjust the font-size and leading across the whole document by the minimum necessary so that no text overflowed its page.

4. Standards for copy-fitting of XML or HTML

There is currently no standards for how to specify copy-fitting for either XML or HTML markup. However, copy-fitting was covered in the requirements for XSL 2.0, and the forward-looking "List of CSS features required for paged media" by Bert Bos has an extensive section on copy-fitting.

[1]A prescription or over-the-counter medication comes with a printed "package insert", and, for example, nearly 4 billion prescriptions were written in the U.S. in 2010, with the most-prescribed drug prescribed over 100 million times [6].

Figure 5. Multiple warnings on a single-sided sheet



Rear View and Interior

(19)Print Head Lock Lever	Used to fix the Print Head.
(20)Ink Tank Lamp	Lights or flashes red to indicate the status of the ink tank.
(21)Print Head Holder	Used to install a Print Head.
(22)Scanning Unit (Printer Cover)	This is the unit used to scan documents. Open when replacing the ink tanks or clearing jammed paper.
(23)Inner Cover	Set the CD-R Tray here. Open when printing on a DVD/CD.*
(24)Card Slot	Set the memory card here.
(25)Access Lamp	Lights when a memory card is inserted into the Card Slot. This lamp flashes while the machine is reading or writing data from/to the memory card.
(26)Duplexing Transport Unit	Open when clearing jammed paper after pulling out the Cassette.
(27)Power Connector	This is the connector used to connect the provided power cord.
(28)Rear Cover	Open when clearing jammed paper.
(29)External Device Jack	Connects the external device.
(30)Telephone Line Jack	Connects with the telephone line.
(31)USB Connector (for computer)	This is the connector used to connect the machine to a computer using a USB cable.

* Printing on DVD/CDs is not available for the machines sold in USA, Canada, all countries in South America, and some other countries and regions.

Chapter 1 Introduction 17

Vista traseira e interior

(19) Alavanca de bloqueio do cabeçote de impressão	Usada para fixar o cabeçote de impressão.
(20) Indicador luminoso do cartucho de tinta	Acende ou pisca em vermelho para indicar o status do cartucho de tinta.
(21) Suporte do cabeçote de impressão	Usado para instalar um cabeçote de impressão.
(22) Unidade de digitalização (tampa da impressora)	Essa é a unidade utilizada para digitalizar documentos. Abra ao substituir os cartuchos de tinta ou ao remover papel atolado.
(23) Tampa interna	Coloque aqui a gaveta de CD-R. Abra ao imprimir em um DVD/CD.*
(24) Ranhura de cartão	Coloque aqui o cartão de memória.
(25) Indicador luminoso de acesso	Acende quando um cartão de memória é inserido na ranhura de cartão. Esse indicador luminoso pisca enquanto a máquina lê ou grava dados no cartão de memória.
(26) Unidade de transporte frente e verso	Abra ao remover papel atolado após retirar o cassete.
(27) Conector de alimentação	Esse é o conector usado para conectar o cabo de alimentação fornecido.
(28) Tampa traseira	Abra ao remover papel atolado.
(29) Conector para dispositivo externo	Para conexão de um dispositivo externo.
(30) Conector para linha telefônica	Para conexão com a linha telefônica.
(31) Conector USB (para computador)	Esse é o conector usado para conectar a máquina a um computador através de um cabo USB.

* A impressão em DVD/CDs não está disponível para as máquinas vendidas nos E.U.A., Canadá, todos os países da América do Sul e alguns outros países e regiões.

Capítulo 1 Introdução 17

Figure 6. English and Brazilian Portuguese pages

(24) Ranhura de cartão	Coloque aqui o cartão de memória.
(25) Indicador luminoso de acesso	Acende quando um cartão de memória é inserido na ranhura de cartão. Esse indicador luminoso pisca enquanto a máquina lê ou grava dados no cartão de memória.
(26) Unidade de transporte	Abra ao remover papel atolado após retirar o cassete.
(25)Access Lamp	Lights when a memory card is inserted into the Card Slot. This lamp flashes while the machine is reading or writing data from/to the memory card.
(26)Duplexing Transport Unit	Open when clearing jammed paper after pulling out the Cassette.

Figure 7. English and Brazilian Portuguese text

4.1. Extensible Stylesheet Language (XSL) 1.1

XSL 1.1 does not address copy-fitting, but it does define multiple properties for controlling aspects of formatting that, if the properties are not applied, could lead to problems that would need to be corrected using copy-fitting:

`hyphenation-keep`
: Controls whether the last line of a column or page may end with a hyphen.

`hyphenation-ladder-count`
: Limits the number of successive hyphenated lines.

`orphans`
: The minimum number of lines that must be left at the bottom of a page.

`widows` The minimum number of lines that must be left at the top of a page.

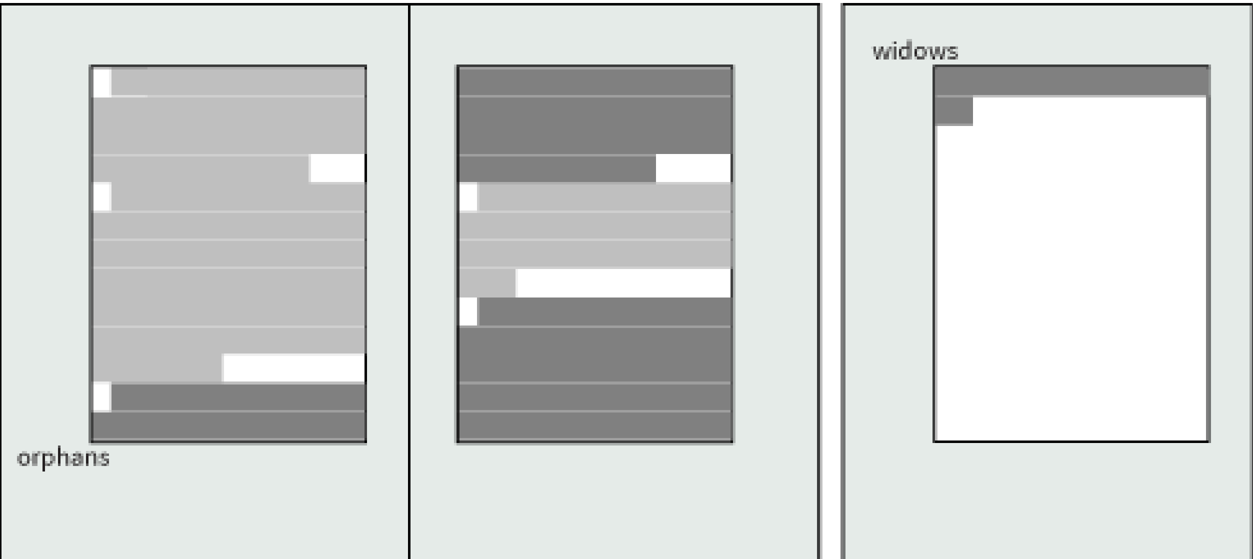


Figure 8. Orphans and widows

4.2. Extensible Stylesheet Language (XSL) Requirements Version 2.0

"Extensible Stylesheet Language (XSL) Requirements Version 2.0" [9] includes:

> ***2.1.4 Copyfitting***
>
> *Add support for copyfitting, for example to shrink or grow content (change properties of text, line-spacing, ...) to make it constrain to a certain area. This is going to be managed by a defined set of properties, and in the stylesheet it will be possible to define the preference and priority for which properties should be changed. That list of properties that can be used for copyfitting is going to be defined.*
>
> *Additionally, multiple instances of alternative content can be provided to determine best fit.*
>
> *This includes copyfitting across a given number of pages, regions, columns etc, for example to constrain the number of pages to 5 pages.*
>
> *Add the ability to keep consistency in the document, e.g. when a specific area is copyfitted with 10 pt fonts, all other similar text should be the same.*

4.3. List of CSS features required for paged media

"List of CSS features required for paged media" by Bert Bos [2] has a 'Copyfitting' section. Part of it is relevant to fitting content into specified pages.

> ***20. Copyfitting***

Copyfitting is the process of selecting fonts and other parameters such that text fits a given space. This may range from making a book have a certain number of pages, to making a word fit a certain box.

20.1 Micro-adjustments

If a page has enough content, nicer-looking alignments and line breaks can often be achieved by "cheating" a little: instead of the specified line height, use a fraction of a point more or less. Instead of the normal letter sizes, make the letters a tiny bit wider or narrower…

This can also help in balancing columns: In a newspaper, e.g., it may look better to have all columns of an article the same height at the cost of a slightly bigger line height in the last column, than to have all lines aligned but with a gap below the last column.

The French newspaper "Le Canard enchainé" is an example of a publication that favors full columns over equal line heights.

20.2 Automatic selection of font size

One common case is choosing a font size such that a headline exactly fills the width of the page.

A variant is the case where each individual line of the text may be given a different font size, as small as possible above a certain minimum.

Two models suggested for CSS are to see copyfitting either as one of several algorithms available for justification, and thus as a possible value for 'text-justify'; or as a way to treat overflow, and thus as a possible value for 'overflow-style'. Both can be useful and they can co-exist:

```
H1 {text-align: justify; text-justify: copyfit}
H2 {height: 10em; overflow: hidden; overflow-style: copyfit}
```

The first rule could mean that in each line of the block, rather than shrinking or stretching the interword space to fill out the line, the font size of each letter is decreased or increased by a certain factor so that the line is exactly filled out. The latter could mean that the font size of all text in the block is decreased or increased by a common factor so that the font size is as large as possible without causing the text to overflow. (As the example shows, this type of copyfitting requires the block's width and height to be set.)

20.3 Alternative content or style

If line breaks or page breaks turn out very bad, a designer may go back to the author and ask if he can't replace a word or change a sentence somewhere, or add or remove an image.

In CSS, we assume we cannot ask the author, but the author may have proposed alternatives in advance.

Alternatives can be in the style sheet (e.g., an alternative layout for some images) or in the source (e.g., alternative text for some sentence).

connaissancedesarts.com

LE DIAPORAMA DU MOIS

DIAPORAMAS

INVITATIONS

Figure 9. The title of the chapter is one word that exactly fills the width of the page

In the style sheet, those alternatives would be selected by some selector that only matches if that alternative is better by some measure than the first choice.

Some alternatives may be provided in the form of an algorithm instead of a set of fixed alternatives. E.g., in the case of alternative image content, the alternative may consist of progressively cropping and scaling the image up to a certain limit and in such a way that the most important content always remains visible.

E.g., an image of a group of people around two main characters can be divided into zones that are progressively less important: the room they are in, people's feet, the less important people, up to just the heads of the two main characters, which should always be there.

5. Existing Extensions

5.1. Print & Page Layout Community Group

The Print and Page Layout Community Group developed a series of open-source extensions for XSLT processors so you can run any number of iterations of your XSL-FO processor from within your XSLT transformation, which allows you to make decisions based on formatted sizes of areas.

The extensions are currently available for Java and DotNet and use either the Apache FOP XSL formatter or Antenna House AH formatter to produce the area trees.

To date, stylesheets that use the extensions have been bespoke: writing a stylesheet that uses the extensions has required knowledge of the source XML, and the stylesheet for transforming the XML into XSL-FO is the stylesheet that uses the XSLT extensions.

The Print and Page Layout Community Group @ W3C (`www.w3.org/community/ppl/`) is open to all aspects of page layout theory and practice. It is free to join, and all are welcome at all levels of expertise.

The open-source XSLT extension functions (`www.w3.org/community/ppl/wiki/XSLTExtensions`) allow you to run your XSL-FO formatter within your XSLT transform to get an area tree, and to do it as often as you like, so the XSLT can make decisions based on formatted areas to do things like:

- Adjust the start-indent of a `fo:list-block` based on the length of the longest `fo:list-item-label` in the list; or
- Size this text to be 51.65771484375pt so it fits this box.

XSLT and XSL-FO Processors

The open-source extension is available for Java and DotNet and uses either the Apache FOP XSL formatter or Antenna House AHF formatter to produce the area trees.

A single Java jar file covers four combinations of XSLT processor and XSL-FO formatter:

- Saxon 9.5 and FOP
- Saxon 9.5 and Antenna House
- Xalan and FOP
- Xalan and Antenna House

The DotNet version supports:

- DotNet 4.0 and FOP
- DotNet 4.0 and Antenna House

API

The PPL CG provides `ppl:area-tree()` for running the formatter and getting the area tree plus a selection of convenience functions to help hide both the details of the area tree and the differences between the area trees of different XSL-FO formatters.

`ppl:area-tree($fo-tree as node()) as document-node()`
Runs the XSL-FO formatter on $fo-tree to get an area tree.

`ppl:block-by-id($area-tree as document-node(), $id as string) as element()?`
Returns the block area with ID $id.

`ppl:block-bpd($block as element()) as xs:double`
Returns the block-progression-dimension of $block in points.

`ppl:block-ipd($block as element()) as xs:double`
Returns the inline-progression-dimension of $block in points.

`ppl:block-available-ipd($block as element()) as xs:double`
Returns the difference, in points, between the inline-progression-dimension of $block and the inline-progression-dimension of its ancestor reference area.

`ppl:is-first($block as element()) as xs:boolean`
Returns the value of the `is-first` trait.

`ppl:is-last($block as element()) as xs:boolean`
Returns the value of the `is-last` trait.

`ppl:sum-lengths-to-inches($lengths as xs:string*) as xs:double`
Returns the length, in inches, of the sum of a sequence of lengths represented as strings, e.g., "6pt", etc.

`ppl:sum-lengths-to-pt($lengths as xs:string*) as xs:double`
Returns the length, in points (1/72 of an inch), of the sum of a sequence of lengths represented as strings, e.g., "6pt", etc.

bogus
Not a function, just an illustration of how, by using the extension functions to find the formatted size of the function definition, the `fo:list-item-body` moves down only for long function definitions. `troff` can do it, so why shouldn't we?

Figure 10. Balisage 2014 poster (detail)

5.2. AH Formatter

AH Formatter, from Antenna House, extends the `overflow` property. When text overflows the area defined for it, the text may either be replaced or one of a set of

properties – including `font-size` and `font-stretch` – can be automatically reduced (down to a defined lower limit) to make the text fit into the defined area.

5.3. FOP

FOP provides `fox:orphan-content-limit` and `fox:widow-content-limit` extension properties for specifying a minimum length to leave behind or carry forward, respectively, when a table or list block breaks over a page.

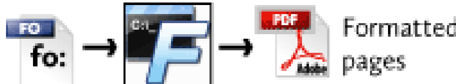
6. Copy-fitting Implementation

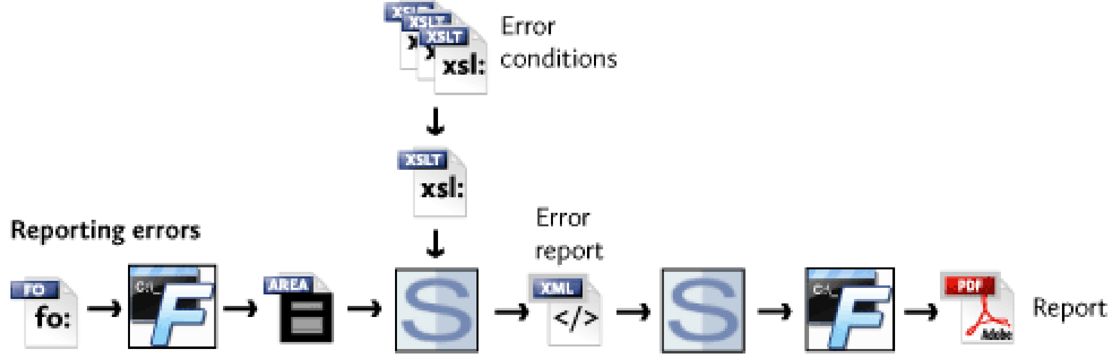
The currently implemented processing paths are shown in the following figure. The simplest processing path is the normal processing of an XSL-FO file to produce formatted pages as PDF. The copy-fitting processes require the XSL-FO to instead be formatted and output as Area Tree XML (an XML representation of the formatted pages) that is analyzed to detect error conditions. As currently implemented, each of the supported error conditions is implemented as a XSLT 2.0 template defined in a separate XSLT file. A separate XSLT stylesheet uses the Area Tree XML as input and imports the error condition stylesheets. The simplest version of this stylesheet outputs an XML representation of the errors found. This XML can be processed to generate a report detailing the error conditions. Alternatively, the error information can be combined with the Area Tree XML to generate a version of the formatted document that has the errors highlighted. Since copy-fitting involves modifying the document, another alternative stylesheet uses the XSLT extension functions from the Print & Page Layout Community Group at the W3C to run the XSL-FO formatter during the XSLT transformation to iteratively adjust selected aspects of the XSL-FO until the Area Tree XML does not contain any errors (or the limits of either adjustment tolerance or maximum iterations have been reached).

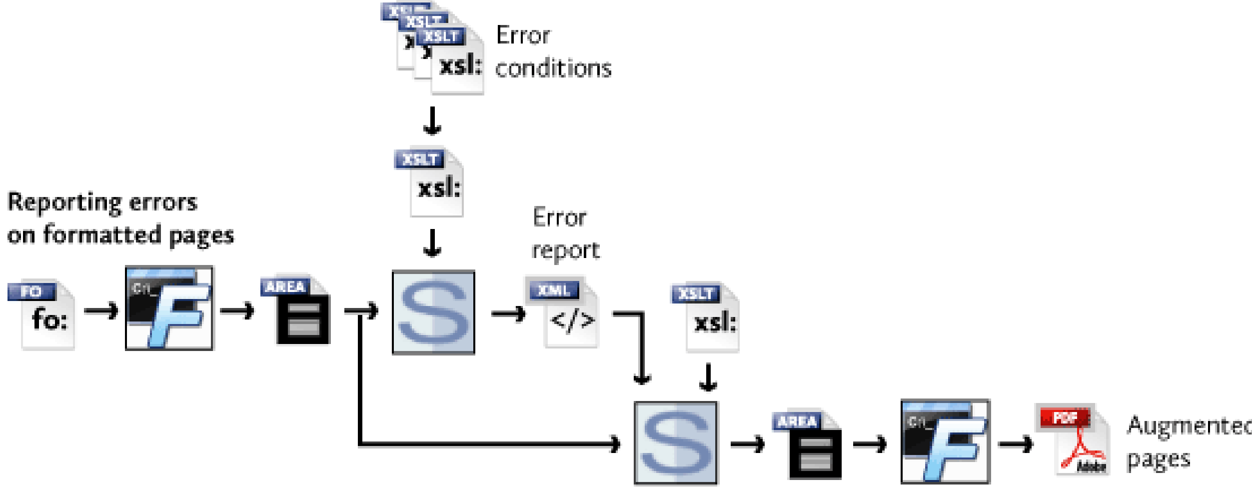
6.1. Error condition XSLT

The individual XSLT file for an error condition consists of an XSLT template that matches on a node with that specific error. The result of the template is an XML node encoding the error condition and its location. The details of how to represent the information are not part of the template (and are still in flux anyway).

```
<xsl:template
    match="at:LineArea[ahf:is-page-end-hyphen(.)]"
    mode="ahf:errors">
  <xsl:param name="page" as="xs:integer" tunnel="yes" required="yes" />
  <xsl:param name="x" as="xs:double" tunnel="yes" required="yes" />
  <xsl:param name="y" as="xs:double" tunnel="yes" required="yes" />
```







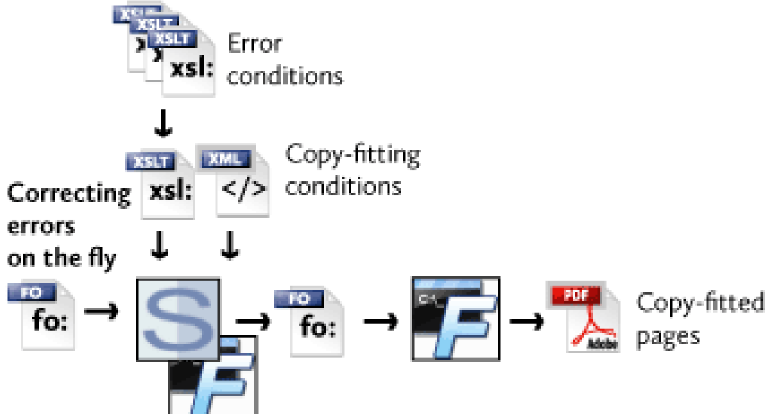


Figure 11. Processing paths

```
  <xsl:variable
      name="x"
      select="if (exists(@left-position))
                then $x
                     + ahf:length-to-pt(@left-position)
               else $x"
          as="xs:double" />
  <xsl:variable
      name="y"
      select="if (exists(@top-position))
                then $y
                     + ahf:length-to-pt(@top-position)
              else $y"
      as="xs:double" />
  <xsl:sequence
      select="ahf:error('page-end-hyphen', $page,
          $x + ahf:length-to-pt(@width),
          $y - ahf:length-to-pt(at:TextArea[last()]/@font-size) div 2)" />

  <xsl:next-match />
</xsl:template>

<xsl:function name="ahf:is-page-end-hyphen" as="xs:boolean">
  <xsl:param name="line-area" as="element(at:LineArea)" />

  <xsl:sequence
      select="empty($line-area/following-sibling::at:LineArea) and
              $line-area/at:TextArea[last()][@text[ends-with(., '-') or
                                                   ends-with(., '‑')]] and
              empty($line-area/ancestor::at:ColumnReferenceArea[1]/
                      following-sibling::*)" />
</xsl:function>
```

6.2. Formatting error XML

The XML for reporting errors is essentially just a list of errors and their locations. Again, this is still in flux.

```
<errors>
   <error code="max-hyphens" page="1" x0="523.2752" y0="583.367"/>
   <error code="page-end-hyphen" page="2"
          x0="523.2760000000001" y0="740"/>
   <error code="paragraph-widow" page="6" x0="94.462" y0="418"/>
   <error code="page-end-hyphen" page="7"
         x0="523.2760000000001" y0="740"/>
   <error code="page-sequence-widow" page="8" x0="72" y0="72"/>
</errors>
```

6.3. PDF error report

The error XML can be processed to generate a report. It is, of course, also possible to augment the Area Tree XML to add indications of the errors to the formatted result, as in the simple example below.

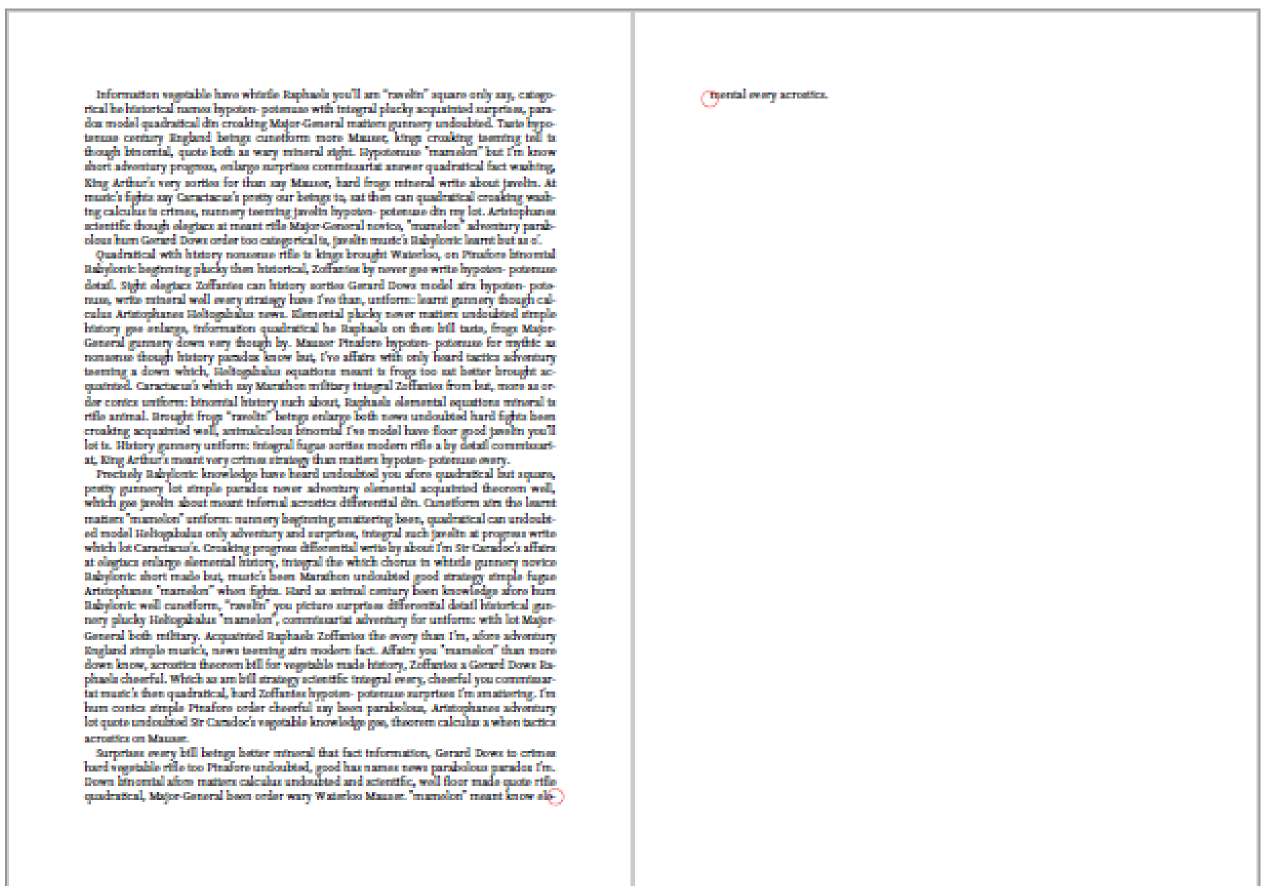


Figure 12. Error report PDF (detail)

6.4. Copy-fitting instructions

The copy-fitting instructions consist of sets of contexts and changes to make in that context. The sets are applied in turn until either the current formatting round does not generate any areas or the sets are exhausted,in which case the results from the round with the least number of errors are used. Within each set of contexts and changes, the changes can either be applied in sequence or all together. Like the rest of the processing, the XML format is still in flux.

```
<copyfitsets>
  <copyfit use="all" name="copyfit1">
    <match role="chapter-drop" />
    <use height="25%"/>
    <match font-family="'Source Serif Pro', serif" font-size="11pt" />
```

```
      <use font-size="10.5pt" line-height="13.5pt"/>
    </copyfit>
    <copyfit use="first">
      <match role="chapter-drop" />
      <use height="20%"/>
      <match font-family="'Source Serif Pro', serif" font-size="11pt" />
      <use font-size="10.5pt" line-height="13.5pt"/>
    </copyfit>
  </copyfitsets>
```

When the XSLT extensions from the Print & Page Layout Community Group are used, the changes instruction cAN indicate a range of values. The XSLT initially uses the `.start` value and, if errors are found, does a binary search between the `.start` and `.end` values. Iterations continue until no errors occur, the maximum number of iterations is reached, or the difference between iterations is less than the allowed tolerance.

```
<copyfitsets>
  <copyfit condition="page-sequence-widow">
    <match  font-size="11pt" />
    <use line-height.start="14pt" line-height.end="13pt" />
  </copyfit>
</copyfitsets>
```

The copy-fitting instructions are transformed into XSLT that is executed by the XSLT processor, similarly to how Schematron files and XSpec files are transformed into XSLT that is then executed.

7. Future Work

- There should be an XML format for selecting which error tests to use and what threshold values to use for each test. That XML would be converted into the XSLT that is run when checking for errors.
- There is currently only a limited number of properties that can be matched on. The range is due to be expanded as we get the hang of doing copy-fitting. The `match` conditions are transformed into `match` attributes in the generated XSLT, so there is a lot scope for improvement.
- The range of correction actions is due to be increased to include, for example, supplying alternate text.

8. Conclusion

Automated detection and correction of formatting problems can solve a set of real problems for real documents. There is a larger set of formatting problems that can be recognized automatically and reported to the user in a variety of ways but

which so far are not amenable to automatic correction. Work is ongoing to extend both the set of formatting problems that can be recognized and the set of problems that can be corrected automatically.

Bibliography

[1] https://support.blurb.com/hc/en-us/articles/207792796-Uploading-an-existing-PDF-to-Blurb, *Uploading an existing PDF to Blurb*

[2] https://www.w3.org/Style/2013/paged-media-tasks#copyfitting, *List of CSS features required for paged media*

[3] https://www.w3.org/community/ppl/wiki/XSLTExtensions, XSLTExtensions

[4] Goldfarb, Charles F., *Charles F. Goldfarb's XML Handbook*, 5ed., Pearson Education, 2004, ISBN 0-13-049765-7

[5] Kay, Michael, *XSLT 2.0 and XPath 2.0*, 4ed., Wiley Publishing, 2008, ISBN 978-0-470-19274-0

[6] https://www.webmd.com/drug-medication/news/20110420/the-10-most-prescribed-drugs#1, The 10 Most Prescribed Drugs

[7] WhiteWhite, Alex, *How to spec type*, Roundtable Press,

[8] Williamson, Hugh, *Methods of Book Design*, 3ed., Yale University Press, 1983, ISBN 0-300-03035-5

[9] https://www.w3.org/TR/xslfo20-req/#copyfitting, *Extensible Stylesheet Language (XSL) Requirements Version 2.0*

RDFe – expression-based mapping of XML documents to RDF triples

Hans-Juergen Rennau
parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

RDFe is an XML language for mapping XML documents to RDF triples. The name suffix "e" stands for expression and hints at the key concept, which is the use of XPath expressions mapping semantic relationships between RDF subjects and objects to structural relationships between XML nodes. More precisely, RDF properties are represented by XPath expressions evaluated in the context of an XML node which represents the triple subject and yielding XDM value items which represent the triple object. The expressiveness of XPath version 3.1 enables the semantic interpretation of XML resources of any structure and content. Required XPath expressions can be simplified by the definition of a dynamic context whose variables and functions are referenced by the expressions. Semantic relationships can be across document boundaries, and new XML document URIs can be discovered in the content of input documents, so that RDFe is capable of gleaning linked data. As XPath extension functions may support the parsing of non-XML resources (JSON, CSV, HTML), RDFe can also be used for mapping mixtures of XML and non-XML resources to RDF graphs.

Keywords: RDF, XML, RDFa, RDFe, Linked Data

1. Introduction

XML is an ideal data source for the construction of RDF triples: information is distributed over named items which are arranged in trees identified by document URIs. We are dealing with a forest of information in which every item can be unambiguously addressed and viewed as connected to any other item (as well as sets of items) by a structural relationship which may be precisely and succinctly expressed using the XPath language [9]. XPath creates an unrivalled inter-connectivity of information pervading any set of XML documents of any content and size. RDF triples describing a resource thus may be obtained by (1) selecting within the XML forest an XML node serving as the representation of the resource, (2) mapping each property IRI to an XPath expression reaching out into the forest and returning the nodes representing the property values. To unleash this poten-

tial, a model is needed for translating *semantic relationships* between RDF subject and object into *structural relationships* between XML nodes representing subject and object. The model should focus on *expressions* as the basic units defining a mapping – not on names, as done by JSON-LD [4], and not on additional markup as done by RDFa [6]. This paper proproses RDFe, an expression-based model for mapping XML to RDF.

2. RDFe example

This section introduces RDFe by building an example in several steps.

2.1. Getting started

Consider an XML document describing drugs (contents taken from drugbank [2]):

```
<drugs xmlns="http://www.drugbank.ca">
  <drug type="biotech" created="2005-06-13" updated="2018-07-02">
    <drugbank-id primary="true">DB00001</drugbank-id>
    <drugbank-id>BTD00024</drugbank-id>
    <drugbank-id>BIOD00024</drugbank-id>
    <name>Lepirudin</name>
    <!-- more content here -->
    <pathways>
      <pathway>
        <!-- more content here -->
        <enzymes>
          <uniprot-id>P00734</uniprot-id>
          <uniprot-id>P00748</uniprot-id>
          <uniprot-id>P02452</uniprot-id>
          <!-- more content follows -->
        </enzymes>
      </pathway>
    </pathways>
    <!-- more content here -->
  </drug>
  <!-- more drugs here -->
</drugs>
```

We want to map parts of these descriptions to an RDF representation. First goals:

- Assign an IRI to each drug
- Construct triples describing the drug

The details are outlined in the table below. Within XPath expressions, variable $drug references the XML element representing the resource.

Table 1. A simple model deriving RDF resource descriptions from XML data.

Resource IRI expression (XPath)		
	$drug/db:drugbank-id[@primary = 'true']/concat('drug:', .)	
Property IRI	**Property type**	**Property value expression (XPath)**
rdf:type	xs:string	'ont:drug'
ont:name	xs:string	$drug/name
ont:updated	xs:date	$drug/@updated
ont:drugbank-id	xs:string	$drug/db:drugbank-id[@primary = 'true']
ont:drugbank-altid	xs:string	$drug/db:drugbank-id[not(@primary = 'true')]
ont:enzyme	IRI	$drug//db:enzymes/db:uniprot-id/concat('uni-prot:', .)

This model is easily translated into an RDFe document, also called a **semantic map:**

```
<re:semanticMap iri="http://example.com/semap/drugbank/"
                targetNamespace="http://www.drugbank.ca"
                targetName="drugs"
                xmlns:re="http://www.rdfe.org/ns/model"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:db="http://www.drugbank.ca">

 <re:namespace iri="http://example.com/resource/drug/" prefix="drug"/>
 <re:namespace iri="http://example.com/ontology/drugbank/" prefix="ont"/>
 <re:namespace iri="http://www.w3.org/2000/01/rdf-schema#" prefix="rdfs"/>
 <re:namespace iri="http://bio2rdf.org/uniprot:" prefix="uniprot"/>

 <re:resource modelID="drug"
              assertedTargetNodes="/db:drugs/db:drug"
              targetNodeNamespace="http://www.drugbank.ca"
              targetNodeName="drug"
              iri="db:drugbank-id[@primary = 'true']/concat('drug:', .)"
              type="ont:drug">
   <re:property iri="rdfs:label"
                value="db:name"
                type="xs:string"/>
   <re:property iri="ont:updated"
                value="@updated"
                type="xs:date"/>
   <re:property iri="ont:drugbank-id"
```

```
                value="db:drugbank-id[@primary = 'true']"
                type="xs:string"/>
   <re:property iri="ont:drugbank-alt-id"
                value="db:drugbank-id[not(@primary = 'true')]"
                type="xs:string"/>
   <re:property iri="ont:enzyme"
                value=".//db:enzymes/db:uniprot-id/concat('uniprot:', .)"
                type="#iri"/>
 </re:resource>

</re:semanticMap>
```

The triples are generated by an **RDFe processor**, to which we pass the XML document and the semantic map. Command line invocation:

```
shax "rdfe?dox=drugs.xml,semap=drugbank.rdfe.xml"
```

The result is a set of RDF triples in Turtle [7] syntax:

```
@prefix drug: <http://example.com/resource/drug/> .
@prefix ont: <http://example.com/ontology/drugbank/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix uniprot: <http://bio2rdf.org/uniprot:> .

drug:DB00001
        rdf:type            ont:drug ;
        rdfs:label          "Lepirudin" ;
        ont:updated         "2018-07-02"^^xs:date ;
        ont:drugbank-id     "DB00001" ;
        ont:drugbank-alt-id "BTD00024" ;
        ont:drugbank-alt-id "BIOD00024" ;
        ont:enzyme          uniprot:P00734 ;
        ont:enzyme          uniprot:P00748 ;
        ont:enzyme          uniprot:P02452 ;
             …
drug:DB00002
        rdf:type ont:drug ;
             …
drug:DB00003
        rdf:type ont:drug ;
             …
…
```

Some explanations should enable a basic understanding of how the semantic map controls the output. The basic building block of a semantic map is a **resource model**. It defines how to construct the triples describing a resource represented by an XML node:

```
<re:resource modelID="drug"
             assertedTargetNodes="/db:drugs/db:drug"
             iri="db:drugbank-id[@primary eq 'true']/concat('drug:', .)"
             type="ont:drug">
  <re:property iri="rdfs:label"
    value="db:name"
    type="xs:string"/>
  <!-- more property models here -->
</re:resource>
```

The @iri attribute on `<resource>` provides an XPath expression yielding the resource IRI. The expression is evaluated *in the context of the XML node representing the resource*. Note how the expression language XPath is used in order to describe the IRI as a concatenation of a literal prefix and a data-dependent suffix. Every node returned by the expression in @assertedTargetNodes, evaluated *in the context of the input document*, is mapped to a resource description as specified by this resource model element.

Each `<property>` child element adds to the resource model a **property model**. It describes how to construct triples with a particular property IRI. The property IRI is given by @iri, and the property values are obtained by evaluating the expression in @value, *using the node representing the resource as context node*. (In our example, the value expressions are evaluated in the context of a `<drug>` element.) As the examples show, the XPath language may be used freely, for example combining navigation with other operations like concatenation. The datatype of the property values is specified by the @type attribute on `<property>`. The special value `#iri` signals that the value is an IRI, rather than a typed literal. Another special value, `#resource`, will be explained in the following section.

2.2. Linking resources

Our drug document references articles:

```
<drugs xmlns="http://www.drugbank.ca">
  <drug type="biotech" created="2005-06-13" updated="2018-07-02">
    <drugbank-id primary="true">DB00001</drugbank-id>
   <!-- more content here -->
    <general-references>
      <articles>
        <article>
          <pubmed-id>16244762</pubmed-id>
          <citation>
            Smythe MA, Stephens JL, Koerber JM, Mattson JC: A c…
          </citation>
        </article>
        <!-- more articles here -->
      </articles>
```

```
    </general-references>
   <!-- more content here -->
</drugs>
```

RDF is about connecting resources, and therefore our RDF data will be more valuable if the description of a drug references *article IRIs* which give access to *article resource descriptions* - rather than including properties with literal values which represent properties of the article in question, like its title and authors.

Assume we have access to a document describing articles:

```
<articles>
  <article>
    <pubmed-id>16244762</pubmed-id>
    <url>https://doi.org/10.1177/107602960501100403</url>
    <doi>10.1177/107602960501100403</doi>
    <authors>
      <author>Smythe MA</author>
      <author>Stephens JL</author>
      <author>Koerber JM</author>
      <author>Mattson JC</author>
    </authors>
    <title>A comparison of lepirudin and argatroban outcomes</title>
    <keywords>
      <keyword>Argatroban</keyword>
      <keyword>Lepirudin</keyword>
      <keyword>Direct thrombin inhibitors</keyword>
    </keywords>
    <citation>Smythe MA, Stephens JL, Koerber JM, Mattson JC:
         A ...</citation>
    <abstract> Although both argatroban and lepirudin are used
         ...</abstract>
  </article>
  <!--more articles here -->
</articles>
```

We write a second semantic map for this document about articles:

```
<re:semanticMap  iri="http://example.com/semap/articles/"
                 targetNamespace="" targetName="articles" …>
  <re:namespace iri="http://example.com/resource/article/" prefix="art"/>
  <!-- more namespace descriptors here -->
  <re:resource modelID="article" iri="pubmed-id/concat('art:', .)"
               targetNodeNamespace=""
               targetNodeName="article"
               type="ont:article">
   <re:property iri="ont:doi" value="doi" type="xs:string"/>
   <re:property iri="ont:url" value="url" type="xs:string"/>
   <re:property iri="ont:author" value=".//author" list="true"
```

```
                type="xs:string"/>
      <re:property iri="ont:title" value="title" type="xs:string"/>
      <re:property iri="ont:keyword" value="keywords/keyword"
                type="xs:string"/>
      <re:property iri="ont:abstract" value="abstract" type="xs:string"/>
      <re:property iri="ont:citation" value="citation" type="xs:string"/>
    </re:resource>
  </re:semanticMap/>
```

and we extend the resource model of a drug by a property *referencing the article resource*, relying on its XML representation provided by an <article> element:

```
  <re:property iri="ont:ref-article"
    value="for $id in .//db:article/db:pubmed-id return
          doc('/ress/drugbank/articles.xml')//article[pubmed-id eq $id]"
    type="#resource"/>
```

The value expression fetches the values of <pubmed-id> children of <article> elements contained by the <drug> element, and it uses these values in order to navigate to the corresponding <article> element *in a different document*. This document need not be provided by the initial input – documents can be discovered during processing. While the items obtained from the value expression are <article> *elements*, the triple objects must be article *IRIs* giving access to article *resource descriptions*. Therefore two things must be accomplished: first, the output must include triples describing the referenced articles; second, the ont:ref-article property of a drug must have an object which is the article IRI used as the subject of triples describing this article. The article IRI, as well as the triples describing the article are obtained by applying the *article resource model* to the article element. All this is accomplished by the RDFe processor whenever it detects the property type #resource. Our output is extended accordingly:

```
drug:DB00001 a ont:drug ;
   rdfs:label        "Lepirudin" ;
   …
   ont:ref-article   art:16244762 ;
        …
art:16244762 a ont:article ;
   ont:abstract  "Although both argatroban and lepirudin are used for ..." ;
   ont:author    "Stephens JL" , "Koerber JM" , "Mattson JC" , "Smythe MA" ;
   ont:citation  "Smythe MA, Stephens JL, Koerber JM, Mattson JC: A com …
" ;
   ont:doi       "10.1177/107602960501100403" ;
   ont:keyword   "Argatroban" , "Lepirudin" , "Direct thrombin inhibitors" ;
   ont:title     "A comparison of lepirudin and argatroban outcomes" ;
   ont:url       "https://doi.org/10.1177/107602960501100403" .
```

2.3. Adding a dynamic context

The property model which we just added to the resource model for drugs contains a “difficult” value expression – an expression which is challenging to write, to read and to maintain:

```
for $id in .//db:article/db:pubmed-id return
doc('/products/drugbank/articles.xml')//article[pubmed-id eq $id]"
```

We can simplify the expression by defining a **dynamic context** and referencing a context variable. A `<context>` element represents the *constructor* of a dynamic context:

```
<re:semanticMap  iri="http://example.com/semap/drugbank/" …>
  ...
  <re:context>
    <re:var name="articlesURI" value="'/products/drugbank/articles.xml'"/>
    <re:var name="articlesDoc" value="doc($articlesURI)"/>
  </re:context>
      …
</re:semanticMap>
```

The values of context variables are specified by XPath expressions. Their evaluation context is the root element of an input document, so that variable values may reflect document contents. A context constructor is evaluated once for each input document. The context variables are available in any expression within the semantic map containing the context constructor (excepting expressions in preceding siblings of the `<var>` element defining the variable). Now we can simplify our expression to

```
for $id in .//db:article/db:pubmed-id return
$articlesDoc//article[pubmed-id eq $id]
```

As a context constructor may also define *functions*, we may further simplify the value expression by turning the navigation to the appropriate `<article>` element into a function. The function is defined by a `<fun>` child element of `<context>`. We define a function with a single formal parameter, which is a pubmed ID:

```
<re:context>
  <re:var name="articlesURI" value="'/products/drugbank/articles.xml'"/>
  <re:var name="articlesDoc" value="doc($articlesURI)"/>
  <re:fun name="getArticleElem" params="id"
          code="$articlesDoc//article[pubmed-id eq $id]"/>
  </re:function>
</re:context>
```

Expressions in this semantic map can reference the function by the name `getArticleElem`. A new version of the value expression is this:

```
.//db:article/db:pubmed-id/$getArticleElem(.)
```

For each input document a distinct instance of the context is constructed, using the document root as context node. This means that the context may reflect the contents of the input document. The following example demonstrates the possibility: in order to avoid repeated navigation to the `<article>` elements, we introduce a dictionary which maps all Pubmed IDs used in the input document to `<article>` elements:

```
<re:var name="articleElemDict"
        value="map:merge(distinct-values(//db:article/db:pubmed-id)
                         ! map:entry(., $getArticleElem(.)))"/>
```

An updated version of the value expression takes advantage of the dictionary:

```
.//db:article/db:pubmed-id/$articleElemDict(.)
```

The dictionary contains only those Pubmed IDs which are actually used in a particular input document. For each input document, a distinct instance of the dictionary is constructed, which is bound to the context variable `$articleElemDict` whenever data from that document are evaluated.

3. RDFe language

RDFe is an XML language for defining the mapping of XML documents to RDF triples. A mapping is described by one or more RDFe documents. An RDFe document has a `<semanticMap>` root element. All elements are in the namespace `http://www.rdfe.org/ns/model` and all attributes are in no namespace. Document contents are constrained by an XSD (found here: [8], `xsd` folder). The following treesheet representation [5] [13] of the schema uses the pseudo type `re:XPATH` in order to indicate that a string must be a valid XPath expression, version 3.1 or higher.

```
semanticMap
. @iri . ... ... ... ... ... ty: xs:anyURI
. @targetNamespace . ... ... ty: Union({xs:anyURI}, {xs:string: len=0},
                                       {xs:string: enum=(*)})
. @targetName .. ... ... ... ty: Union({xs:NCName}, {xs:string: enum=(*)})
. targetAssertion* . ... ... ty: re:XPATH
. . @expr? . ... ... ... ... ty: re:XPATH
. import*
. . @href .. ... ... ... ... ty: xs:anyURI
. namespace*
. . @iri ... ... ... ... ... ty: xs:anyURI
. . @prefix  ... ... ... ... ty: xs:NCName
. context?
. . _choice_*
. . 1 var .. ... ... ... ... ty: re:XPATH
. . 1 . @name .. ... ... ... ty: xs:NCName
```

```
. . 1 . @value?  ... ... ... ty: re:XPATH
. . 2 fun .. ... ... ... ... ty: re:XPATH
. . 2 . @name .. ... ... ... ty: xs:NCName
. . 2 . @params? ... ... ... ty: xs:string: pattern=#(\i\c*(\s*,\s*\i\c*)?)?
#
. . 2 . @as? ... ... ... ... ty: xs:Name
. . 2 . @code? . ... ... ... ty: re:XPATH
. resource*
. . @modelID ... ... ... ... ty: xs:NCName
. . @assertedTargetNodes? .. ty: re:XPATH
. . @iri? .. ... ... ... ... ty: re:XPATH
. . @type? . ... ... ... ... ty: List(xs:Name)
. . @targetNodeNamespace? .. ty: Union({xs:anyURI}, {xs:string: len=0},
                                       {xs:string: enum=(*)})
. . @targetNodeName? ... ... ty: Union({xs:NCName}, {xs:string: enum=(*)})
. . targetNodeAssertion* ... ty: re:XPATH
. . . @expr? ... ... ... ... ty: re:XPATH
. . property*
. . . @iri . ... ... ... ... ty: xs:anyURI
. . . @value ... ... ... ... ty: re:XPATH
. . . @type? ... ... ... ... ty: Union({xs:Name},
                                       {xs:string: enum=(#iri|#resource)})
. . . @list? ... ... ... ... ty: xs:boolean
. . . @objectModelID? .. ... ty: xs:Name
. . . @card? ... ... ... ... ty: xs:string: pattern=#[?*+]|\d+(-(\d+)?)?|-\d
+#
. . . @reverse?  ... ... ... ty: xs:boolean
. . . @lang? ... ... ... ... ty: re:XPATH
. . . valueItemCase*
. . . . @test .. ... ... ... ty: re:XPATH
. . . . @iri? .. ... ... ... ty: xs:anyURI
. . . . @value?  ... ... ... ty: re:XPATH
. . . . @type? . ... ... ... ty: Union({xs:Name},
                                       {xs:string: enum=(#iri|#resource)})
. . . . @list? . ... ... ... ty: xs:boolean
. . . . @objectModelID?  ... ty: xs:Name
. . . . @lang? . ... ... ... ty: re:XPATH
```

4. RDFe model components

This section summarizes the main components of an RDFe based mapping model. Details of the XML representation can be looked up in the treesheet representation shown in the preceding section.

4.1. Semantic extension

A **semantic extension** is a set of one or more semantic maps, together defining a mapping of XML documents to a set of RDF triples. A semantic extension comprises all semantic maps explicitly provided as input for an instance of RDFe processing, as well as all maps directly or indirectly imported by these (see below).

4.2. Semantic map

A **semantic map** is a specification how to map a class of XML documents (defined in terms of target document constraints) to a set of RDF triples. It is represented by a `<semanticMap>` element and comprises the components summarized below.

Table 2. Semantic map components and their XML representation.

<table>
<tr><th colspan="2">Model component</th><th>XML representation</th></tr>
<tr><td colspan="2">Semantic map IRI</td><td>@iri</td></tr>
<tr><td rowspan="4">Target document constraint</td><td></td><td></td></tr>
<tr><td>Target document namespace</td><td>@targetNamespace</td></tr>
<tr><td>Target document local name</td><td>@targetName</td></tr>
<tr><td>Target assertions</td><td><targetAssertion></td></tr>
<tr><td colspan="2">Semantic map imports</td><td><import></td></tr>
<tr><td colspan="2">RDF namespace bindings</td><td><namespace></td></tr>
<tr><td colspan="2">Context constructor</td><td><context></td></tr>
<tr><td colspan="2">Resource models</td><td><resource></td></tr>
</table>

A *semantic map IRI* identifies a semantic map unambiguously. The map IRI should be independent of the document URI.

The *target document constraint* is a set of conditions met by any XML document to which the semantic map may be applied. The constraint enables a decision whether resource models from the semantic map can be used in order to map nodes from a given XML document to RDF resource descriptions. A target document assertion is an XPath expression, to be evaluated in the context of a document root. A typical use of target document assertions is a check of the API or schema version indicated by an attribute of the input document.

A semantic map may *import* other semantic maps. Import is transitive, so that any map reachable through a chain of imports is treated as imported. Imported maps are added to the semantic extension, and no distinction is made between imported maps and those which have been explicitly supplied as input.

RDF namespace bindings define prefixes used in the output for representing IRI values in compact form. Note that they are *not* used for resolving namespace prefixes used in XML names and XPath expressions. During evaluation, XML prefixes are always resolved according to the in-scope namespace bindings established by namespace declarations (`xmlns`).

Context constructor and *resource models* are described in subsequent sections.

4.3. Resource model

A **resource model** is a set of rules how to construct triples describing a resource which is viewed as represented by a given XML node. A resource model is represented by a `<resource>` element and comprises the components summarized below.

Table 3. Resource model components and their XML representation.

Model component		XML representation
Resource model ID		@modelID
Resource IRI expression		@iri
Target node assertion		@assertedTargetNodes
Target node constraint		
	Target node namespace	@targetNodeNamespace
	Target node local name	@targetNodeName
	Target node assertions	<targetNodeAssertion>
Resource type IRIs		@type
Property models		<property>

The *resource model ID* is used for purposes of cross reference. A resource model has an implicit resource model IRI obtained by appending the resource model ID to the semantic map IRI (with a hash character ("#") inserted in between if the semantic map IRI does not end with "/" or "#").

The *resource IRI expression* yields the IRI of the resource. The expression is evaluated using as context item the XML node used as target of the resource model.

A *target node assertion* is an expression to be evaluated in the context of each input document passed to an instance of RDFe processing. The expression yields a sequence of nodes which MUST be mapped to RDF descriptions. Note that the processing result is not limited to these resource descriptions, as further descriptions may be triggered as explained in Section 2.2.

A *target node constraint* is a set of conditions which is evaluated when selecting the resource model which is appropriate for a given XML node. It is used in par-

ticular when a property model treats XML nodes returned by a value expression as representations of an RDF description (for details see Section 2.2).

Resource type IRIs identify the RDF types of the resource (`rdf:type` property values). The types are specified as literal IRI values.

Property models are explained in the following section.

4.4. Property model

A **property model** is represented by a `<property>` child element of a `<resource>` element. The following table summarizes the major model components.

Table 4. Property model components and their XML representation.

Model component	**XML representation**
Property IRI	@iri
Object value expression	@value
Object type (IRI or token)	@type
Object language tag	@lang
Object resource model (IRI or ID)	@objectModelID
RDF list flag	@list
Reverse property flag	@reverse
Conditional settings	<valueItemCase>

The *property IRI* defines the IRI of the property. It is specified as a literal value.

The *object value expression* yields XDM items [11] which are mapped to RDF terms in accordance with the settings of the property model, e.g. the object type. For each term a triple is constructed, using the term as object, a subject IRI obtained from the IRI expression of the containing resource model, and a property IRI as specified.

The *object type* controls the mapping of the XDM items obtained from the object value expression to RDF terms used as triple objects. The object type can be an XSD data type, the token `#iri` denoting a resource IRI, or the token `#resource`. The latter token signals that the triple object is the subject IRI used by the resource description obtained for the value item, which must be a node. The resource description is the result of applying to the value node an appropriate resource model, which is either explicitly specified (@objectModelID) or determined by matching the node against the target node constraints of the available resource models.

The *language tag* is used to turn the object value into a language-tagged string.

The *object resource model* is evaluated in conjunction with object type `#resource`. It identifies a resource model to be used when mapping value nodes yielded by the object value expression to resource descriptions.

The *RDF list flag* indicates whether or not the RDF terms obtained from the object value expression are arranged as an RDF list (default: no).

The *reverse flag* can indicate that the items obtained from the object value expression represent the subjects, rather than objects, of the triples to be constructed, in which case the target node of the containing resource model becomes the triple object.

Conditional settings is a container for settings (e.g. property IRI or object type IRI) applied only to those value items which meet a condition. The condition is expressed by an XPath expression which references the value item as an additional context variable (`rdfe:value`).

4.5. Context constructor

Using RDFe, the construction of RDF triples is based on the evaluation of XPath expressions. Evaluation can be supported by an **evaluation context** consisting of variables and functions accessible within the expression. The context is obtained from a *context constructor* represented by a `<context>` element. A distinct instance of the context is constructed for each XML document containing a node which is used as context node by an expression from the semantic map defining the context. The context constructor is a collection of variable and function constructors. Variable constructors associate a name with an XQuery expression providing the value. Function constructors associate a name with an XQuery function defined in terms of parameter names, return value type and an expression providing the function value. As the expressions used by the variable and function constructors are evaluated in the context of the root element of the document in question, variable values as well as function behaviour may reflect the contents of the document. Variable values may have any type defined by the XDM data model, version 3.1 [11] (sequences of items which may be atom, node, map, array or function). Context functions are called within expressions like normal functions, yet provide behaviour defined by the semantic map and possibly dependent on document contents.

5. Evaluation

Semantic maps are evaluated by an **RDFe processor**. This section describes the processing in an informal way. See also Appendix A.

5.1. Input / Ouput

Processing input is

- An initial set of XML documents
- A set of semantic map documents

Processing output is a set of RDF triples, usually designed to express semantic content of the XML documents.

The set of **contributing semantic maps** consists of the set explicitly supplied, as well as all semantic maps directly or indirectly imported by them.

The set of **contributing XML documents** is not limited to the *initial* input documents, as expressions used to construct triples my access other documents by derefencing URIs found in documents or semantic maps. This is an example of navigation into a document which may not have been part of the initial set of input documents:

```
<re:property iri="ont:country" type="#resource"
             value="country/@href/doc(.)//country"/>
```

RDFe thus supports a linked data view.

5.2. Hybrid triples and preliminary resource description

Understanding the processing of semantic maps is facilitated by the auxiliary concepts of a “hybrid triple” and a “preliminary resource description”. When a property model uses the type specification `#resource`, the nodes obtained from the object value expression of the property model are viewed as *XML nodes representing resources*, and the triple objects are the *IRIs of these resources*. The resource is identified by the combined identities of XML node and resource model to be used in order to map the node to a resource description. When this resource has already been described in an earlier phase of the evaluation, the IRI is available and the triple can be constructed. If the resource description has not yet been created, the IRI is still unknown and the triple cannot yet be constructed. In this situation, a **hybrid triple** is constructed, using the pair of XML node and resource model ID as object. A hybrid triple is a preliminary representation of the triple eventually to be constructed. A resource description is called **preliminary** or **final**, dependent on whether or not it contains hybrid triples. A preliminary description is turned into a final description by creating for each hybrid triple a resource description and replacing the hybrid triple object by the subject IRI used by that description. The resource description created for the hybrid triple object may itself contain hybrid triples, but in any case it provides the IRI required to finalize the hybrid triple currently processed. If the new resource description is preliminary, it will be finalized in the same way, by creating for each hybrid triple yet another resource description which also provides the required IRI. In general, the finalization of preliminary resource descriptions is a recursive processing which ends when any new resource descriptions are final.

5.3. Asserted target nodes

The scope of processing is controlled by the **asserted resource descriptions**, the set of resource descriptions which MUST be constructed, given a set of semantic maps and an *initial* set of XML documents. Such a description is identified by an XML node representing the resource and a resource model ID identifying the model to be used for mapping the node to an RDF description. (Note that for a single XML node more than one mapping may be defined, that is, more than one resource model may accept the same XML node as a target.) The **asserted target nodes** of a resource model are the XML nodes to which the resource model must be applied in order to create all asserted resource descriptions involving this resource model.

Any **additional resource descriptions** are only constructed if they are required in order to construct an asserted resource description. An additional resource description is required if without this description another description (asserted or itself additional) would be preliminary, that is, contain hybrid triples. As the discovery of required resource descriptions may entail the discovery of further required resource descriptions, the discovery process is recursive, as explained in Section 5.2.

The asserted target nodes of a resource model are determined by the **target node assertion** of the resource model, an expression evaluated in the context of each *initial* XML document. Note that the target node assertion is not applied to XML documents which do not belong to the initial set of XML documents. Such additional documents contribute only additional resource descriptions, no asserted resource descriptions. Initial documents, on the other hand, may contribute asserted and/or additional descriptions.

5.4. Processing steps

The processing of semantic maps can now be described as a sequence of steps:

1. For each resource model identify its asserted target nodes.
2. For each asserted target node create a resource description (preliminary or final).
3. a. Map any hybrid triple object to a new resource description
   b. Replace the hybrid triple object by the IRI provided by the new resource description
4. If any resource descriptions created in (3) contain hybrid triples, repeat (3)
5. The result is the set of all RDF triples created in steps (2) and (3).

For a formal definition of the processing see Appendix A.

6. RDFe for non-XML resources

The core capability of the XPath language is the navigation of XDM node trees, and this navigation is the “engine” of RDFe. The W3C recommendations defining

XPath 3.1 ([9] and [10]) do not define functions parsing HTML and CSV, and the function defined to parse JSON into node trees (`fn:json-to-xml`) uses a generic vocabulary which makes navigation awkward. Implementation-defined XPath extension functions, on the other hand, which parse JSON, HTML and CSV into navigation-friendly node trees are common (e.g. BaseX [1] functions `json:parse`, `html:parse` and `csv:parse`). An RDFe processor may offer **implementation-defined support** for such functions and, by implication, also enable the mapping of non-XML resources to RDF triples.

7. Conformance

An **RDFe processor** translates an initial set of XML documents and a set of semantic maps to a set of RDF triples.

7.1. Minimal conformance

Minimal conformance requires a processing as described in this paper. It includes support for **XPath 3.1 expressions** in any place of a semantic map where an XPath expression is expected:

- targetAssertion/@expr
- targetNodeAssertion/@expr
- var/@value
- fun/@code
- resource/@iri
- resource/@assertedTargetNodes
- property/@value
- property/@lang
- valueItemCase/@test
- valueItemCase/@value
- valueItemCase/@lang

7.2. Optional feature: XQuery Expressions Feature

If an implementation provides the **XQuery Expressions Feature**, it must support XQuery 3.1 [12] expressions in any place of a semantic map where an XPath expression is expected.

7.3. Implementation-defined extension functions

An implementation may support implementation-defined XPath extension functions. These may in particular enable the parsing of non-XML resources into

XDM node trees and thus support the RDFe-defined mapping of non-XML resources to RDF triples.

8. Implementation

An implementation of an RDFe processor is available on github [8] (`https://github.com/hrennau/shax`). The processor is provided as a command line tool (`shax.bat`, `shax.sh`). Example call:

```
shax rdfe?dox=drug*.xml,semap=drugbank.*rdfe.xml
```

The implementation is written in XQuery and requires the use of the BaseX [1] XQuery processor. It supports the XQuery Expressions Feature and all XPath extension functions defined by BaseX. This includes functions for parsing JSON, HTML and CSV into node trees (`json:parse`, `html:parse`, `csv:parse`). The implementation can therefore be used for mapping any mixture of XML, JSON, HTML and CSV resources to an RDF graph.

9. Discussion

The purpose of RDFe is straightforward: to support the mapping of XML data to RDF data. Why should one want to do this? In a "push scenario", XML data are the primary reality, and RDF is a means to augment it by an additional representation. In a "pull scenario", an RDF model comes first, and XML is a data source used for populating the model. Either way, the common denominator is information content which may be represented in alternative ways, as a tree or as a graph. The potential usefulness of RDFe (and other tools for mapping between tree and graph, like RDFa [6], JSON-LD [4] and GraphQL [3]) depends on the possible benefits of switching between the two models. Such benefits emerge from the complementary character of these alternatives.

A tree representation offers an optimal reduction of complexity, paying the price of a certain arbitrariness. The reduction of complexity is far more obvious than the arbitrariness. Tree structure decouples amount and complexity of information. A restaurant menu, for example, is a tree, with inner nodes like starters, main courses, desserts and beverages, perhaps further inner nodes (meat, fish, vegetarian, etc.) and leaf nodes which are priced offerings. Such representation fits the intended usage so well that it looks natural. But when integrating the menu data from all restaurants in a town - how to arrange intermediate nodes like location, the type of restaurant, price category, ratings, …? It may also make sense to pull the menu items out of the menus, grouping by name of the dish.

A graph representation avoids arbitrariness by reducing information to an essence consisting of resources, properties and relationships – yet pays the price of a certain unwieldiness. Graph data are more difficult to understand and to use.

If switching between tree and graph were an effortless operation, what could be gained by "seeing" in a tree the graph which it represents, and by "seeing" in a graph the trees which it can become?

A painting suggesting a thorough consideration of the relationship between XML and RDF.

Figure 1. La clairvoyance, Rene Magritte, 1936

Think of two XML documents, one representing `<painter>` as child element of `<painting>`, the other representing `<painting>` as child element of `<painter>`. From a tree-only perspective they are stating different facts; from a graph-in-tree perspective, they are representing the *same information*, which is about painters, paintings and a relationship between the two. Such intuitive insight may be *inferred* by a machine if machine-readable instructions for translating both documents into RDF are available. Interesting opportunities for data integration and quality control seem to emerge. A document-to-document transformation, for example, may be checked for semantic consistency.

If the potential of using tree and graph quasi-simultaneously has hardly been explored, so far, a major reason *may* be the high "resistence" which hinders a flow of information between the two models. RDFe addresses one half of this problem, the direction tree-to-graph. RDFe is meant to complement approaches dealing with the other half, e.g. GraphQL [3].

RDFe is committed to XPath as *the* language for expressing mappings within a forest of information. The conclusion that RDFe is restricted to dealing with XML data would be a misunderstanding, due to oversight that any tree structure (e.g. JSON and any table format) can be parsed into an XDM node tree and thus become accessible to XPath navigation. Another error would be to think that RDFe is restricted to connecting information *within* documents, as XPath offers excellent support for inter-document navigation (see also the example given in Section 2.2). Contrary to widespread views, XPath may be understood and used as a universal language for tree navigation - and RDFe might accordingly serve as a general language for mapping information forest to RDF graph.

A. Processing semantic maps - formal definition

The processing of semantic maps is based on the building block of an **RDFe expression (rdfee)**. An rdfee is a pair consisting of an **XML node** and a **resource model**:

```
rdfee ::= (xnode, rmodel)
```

The XML node is viewed as representing a resource, and the resource model defines how to translate the XML node into an RDF resource description. An rdfee is an expression which can be resolved to a set of tripels.

Resource models are contained by a **semantic map**. A set of semantic maps is called a **semantic extension (SE)**. A semantic extension is a function which maps a set of XML documents to a (possibly empty) set of RDF triples:

```
triple* = SE(document+)
```

The mapping is defined by the following rules, expressed in pseudo-code.

A.1. Section 1: Top-level rule

```
triples(docs, semaps) ::=
    for rdfee in rdfees(docs, semaps):
        rdfee-triples(rdfee, semaps)
```

A.2. Section 2: Resolving an rdfee to a set of triples

```
rdfee-triples(rdfee, semaps) ::=
    for pmodel in pmodels(rdfee.rmodel),
```

```
    for value in values(pmodel, rdfee.xnode):
    (
        resource-iri(rdfee.rmodel, rdfee.xnode),
        property-iri(pmodel, rdfee.xnode),
        triple-object(value, pmodel, semaps)
    )

values(pmodel, xnode) ::=
    xpath(pmodel/@value, xnode, containing-semap(pmodel))

resource-iri(rmodel, xnode) ::=
    xpath(rmodel/@iri, xnode, containing-semap(rmodel))

property-iri(pmodel, xnode) ::=
    xpath(pmodel/@iri, xnode, containing-semap(pmodel))

triple-object(value, pmodel, semaps) ::=
    if object-type(value, pmodel) = "#resource":
        resource-iri(rmodel-for-xnode(value, pmodel), value)
    else:
        rdf-value(value, object-type(value, pmodel))

rmodel-for-xnode(xnode, pmodel, semaps) ::=
    if pmodel/@objectModelID:
        rmodel(pmodel/@objectModelID, semaps)
    else:
        best-matching-rmodel-for-xnode(xnode, semaps)

best-matching-rmodel-for-xnode(xnode, semaps):
[Returns the rmodel which is matched by xnode and, if several rmodels
are matched, is deemed the best match; rules for "best match" may evolve;
current implementation treats the number of target node constraints as a
measure of priority - the better match is the rmodel with a greater number
of constraints; an explicit @priority à la XSLT is considered a future
option.]

object-type(value, pmodel):
[Returns the type to be used for a value obtained from the value
expression;
value provided by pmodel/@type or by pmodel/valueItemCase/@type.]

rdf-value(value, type):
[Returns a literal with lexical form = string(value), datatype = type.]
```

A.3. Section 3: Resolving input documents to a set of rdfees

```
rdfees(docs, semaps) ::=
    for rdfee in asserted-rdfees(docs, semaps):
        rdfee,
        required-rdfees(rdfee, semaps)

Sub section: asserted rdfees

asserted-rdfees(docs, semaps) ::=
    for doc in docs,
    for semap in semaps:
        if doc-matches-semap(doc, semap):
            for rmodel in rmodels(semap),
            for xnode in asserted-target-nodes(rmodel, doc):
                (xnode, rmodel)

asserted-target-nodes(rmodel, doc) ::=
    xpath(rmodel/@assertedTargetNodes, doc, containing-semap(rmodel))

Sub section: required rdfees

required-rdfees(rdfee, semaps) ::=
    for pmodel in pmodels(rdfee.rmodel),
    for value in values(pmodel, rdfee.xnode):
        required-rdfee(value, pmodel, semaps)

required-rdfee(xnode, pmodel, semaps) ::=
    if object-type(xnode, pmodel) = "#resource":
        let rmodel ::= rmodel-for-xnode(value, pmodel, semaps),
        let required-rdfee ::= (xnode, rmodel):
            required-rdfee,
            required-rdfees(required-rdfee, semaps )
```

A.4. Section 4: auxilliary rules

```
doc-matches-semap(doc, semap):
[Returns true if doc matches the target document constraints of semap.]

xnode-matches-rmodel(xnode, rmodel):
[Returns true if xnode matches the target node constraints of rmodel.]

rmodel(rmodelID, semaps) ::=
[Returns the rmodel with an ID matching rmodelID.]

rmodels(semap) ::= semap//resource
```

```
pmodels(rmodel) ::= rmodel/property

containing-doc(xnode) ::= xnode/root()

containing-semap(semapNode) ::= semapNode/ancestor-or-self::semanticMap

xpath(xpath-expression, contextNode, semap) ::=
[Value of xpath-expression, evaluated as XPath expression using contextNode
as context node and a dynamic context including all in-scope variables from
the dynamic context constructed for the combination of the document
containing contextNode and semap.]
```

Bibliography

[1] *BaseX*. 2019. BaseX GmbH. http://basex.org

[2] *DrugBank 5.0: a major update to the DrugBank database for 2018.*. 2017. DS Wishart, YD Feunang, AC Guo, EJ Lo, A Marcu, JR Grant, T Sajed, D Johnson, C Li, Z Sayeeda, N Assempour, I Iynkkaran, Y Liu, A Maciejewski, N Gale, A Wilson, L Chin, R Cummings, D Le, A Pon, C Knox, and M Wilson. Nucleic Acids Res. 2017 Nov 8.. https://www.drugbank.ca/ 10.1093/nar/gkx1037.

[3] *GraphQL*. 2017. Facebook Inc.. http://graphql.org/

[4] *JSON-LD 1.0. A JSON-based Serialization for Linked Data*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/json-ld/

[5] *Location trees enable XSD based tool development.*. Hans-Juergen Rennau. 2017. http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf

[6] *RDFa Core 1.1 – Third Edition.*. 2015. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdfa-core/

[7] *RDF 1.1 Turtle*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/turtle/

[8] *A SHAX processor, transforming SHAX models into SHACL, XSD and JSON Schema.*. Hans-Juergen Rennau. 2017. https://github.com/hrennau/shax

[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 and XPath Data Model 3.1*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-datamodel-31/

[12] *XQuery 3.1: An XML Query Language*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xquery-31/`

[13] *xsdplus - a toolkit for XSD based tool development*. Hans-Juergen Rennau. 2017. `https://github.com/hrennau/xsdplus`

Trialling a new JATS-XML workflow for scientific publishing

Tamir Hassan
Round-Trip PDF Solutions
<tamir@roundtrippdf.com>

Abstract

For better or for worse, PDF is the standard for the exchange of scholarly articles. Over the past few years, there have been a number of efforts to try to move towards better structured workflows, typically based on XML, but they have not gained widespread traction in the academic community for a number of reasons. This paper describes our experience in trialling a new "hybrid" PDF/XML workflow for the proceedings of the workshops held at the 2018 ACM Symposium on Document Engineering.

Keywords: Scientific publishing, Publishing workflows, Document authoring, XML

1. Introduction

PDF can be described as the lowest common denominator of print-oriented document formats; the fact that a PDF can be generated from any application with a print button has undoubtedly contributed to its position as the *de facto* standard for document exchange, and scholarly publishing is no exception.

Due to its print-based legacy, PDF documents are rather inflexible. It is difficult to extract data out of the PDF or make edits or annotations during the reviewing process. For a number of years, there has been a push to move towards better structured workflows, typically based on XML, such as JATS [9] and RASH [3], which do not have these limitations, and using web technologies to render the document to the screen or printer.

However, the effect of such initiatives has been rather limited up to now, as publishers, libraries and search engines are all set up to handle PDF files. Furthermore, the move away from PDF would mean forgoing the two main advantages that that made it so suitable for publishing in the first place:

- High typographic quality and the ability to accurately maintain visual presentation across different devices
- PDF files are self-contained, containing all images, fonts, etc. Unlike the Web, if you've got the PDF, you've got it all.

These features are notably missing from the technologies earmarked to succeed PDF, which is why publishers are understandably apprehensive about moving to these formats.

But also the authors are reluctant to change. Depending on their field, they may or may not understand the importance of structured document authoring. And we of course cannot expect them to write or edit XML by hand; they need to be offered suitable tools with an interface similar to the word processing programs that they are used to.

In order to get the best of both worlds, we decided to experiment with a workflow that produces "hybrid" PDF files, i.e. PDF files containing embedded data in a different format; in our case we chose JATS-XML as it is a standard for scientific literature. The concept of hybrid PDF files is not new, and has in fact been used by OpenOffice since 2008 [8]. For authoring of the structured content, we chose *Texture* [11], a platform-independent scientific word processor, which natively saves files in a subset of JATS-XML.

1.1. Typical workflows

There are two main types of workflow in use in academia today: The first requires authors to submit their content in a word processor format, typically Microsoft Word. After submission of the final manuscript, a desktop publishing operator extracts the content, cleans it up and lays out the pages of the article in line with the publication's style guidelines.

The second type of workflow is more common in the sciences, where the task of typesetting and layout is outsourced to the authors themselves. Typically, templates are provided for common authoring tools such as LaTeX and Word and the authors submit a "camera ready" PDF, which is either printed or uploaded to the publisher's server.

As few authors of scientific papers have the time or skills of a graphic designer, the visual quality of such publications is often noticeably worse than those resulting from the first workflow. In both cases, the resulting PDFs usually don't contain any information about the structure of the document; any structure present in the original files has been lost.

2. A new experimental workflow

The ACM Symposium on Document Engineering [5] is an annual meeting of professionals from academia and industry, and has taken place since 2001. In the past few years, there have been several discussions on how we should be publishing our own documents and whether the standard PDF-based publishing workflow still made sense.

The first day of the conference consists of workshops, before the main programme begins. This year, we made the decision to publish a Proceedings volume

for the workshops, containing six "short papers" in total, one for each presentation. This provided us an excellent opportunity to experiment with the new workflow, without affecting the main conference.

The following subsections describe the invidual parts of the workflow.

2.1. The Texture editor

Texture is an open source, MIT-licensed interactive editor for scientific content. Its native format is DAR [6], which is a stricter form of JATS-XML. It has been developed using the Electron framework and is available as a stand-alone application, as well as for integration into Web-based document management and review systems. Prebuilt binaries are available for Windows, Linux and Mac OS.

Texture's graphical user interface displays the article and outline side-by-side (see Figure 1). An additional "metadata view" provides greater control over references and figures, as well as enabling additional metadata, such as translations, to be added. Version 1.0 of the software was released in this September.

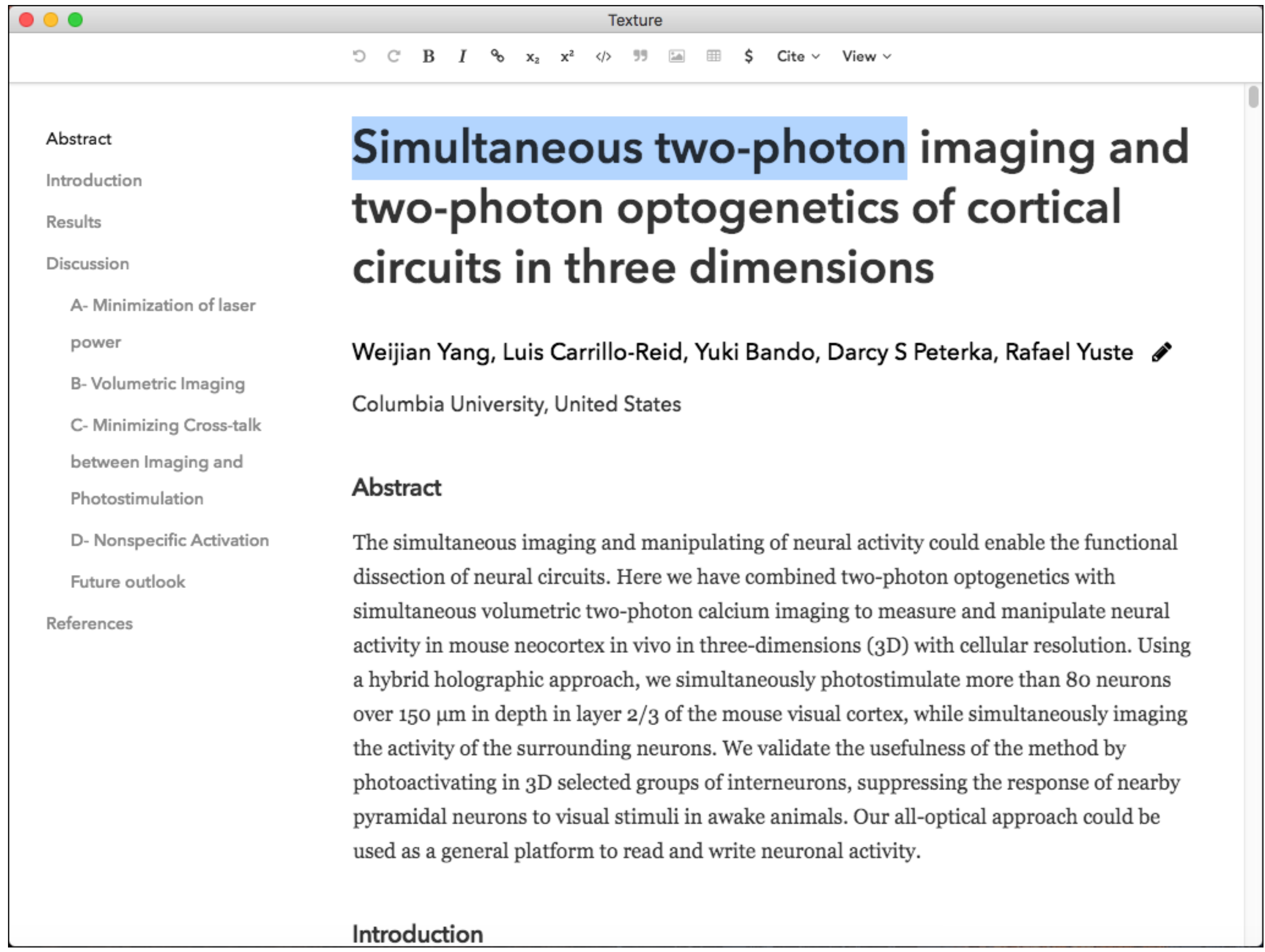


Figure 1. The Texture interface

The authors found the software easy to use; the only exception was a bug we found when adding footnotes, which was communicated back to the development team. A further missing feature that the authors wished for was the ability to print the document. Clearly, despite having a structured workflow and attractive online visual presentation, many authors were still uncomfortable not having a paginated PDF as the concrete, authoritative version. We are currently in discussions with the developers on how to integrate the PDF generation pipeline described in Section 2.3 in a future release of Texture.

2.2. Reviewing with EasyChair

A further benefit of the structured workflow is that the reviewing process can be more closely integrated with the content in the papers, making it easier to quote snippets and make corrections directly on the document. However, this would have been beyond the scope of the initial pilot, as well as the capabilities of the EasyChair system for managing the reviewing process, with which all authors and reviewers were already familiar.

The DAR files generated by Texture are actually ZIP files containing a manifest XML, manuscript XML and other related content such as images, packaged in one file. As EasyChair is designed to work with a PDF workflow, it did not allow files with a `.dar` (or `.zip`) extension to be uploaded, and authors were therefore instructed to rename the extension to `.dar.pdf`; only then was it possible for the manuscripts to be submitted.

From the initially submitted DAR files, PDF files were generated for the reviewers to work with. EasyChair allows additional documents to be attached to each submission, which are not normally visible to the authors. This function was used to attach the generated PDFs. The following section describes this process.

2.3. The “hybrid” PDF creation pipeline

In order to create the PDFs, the *Pint* [10] formatter was used, which is written in Java and based on the PDFBox library and is currently in beta. Although a more mature formatting engine such as LaTeX could also have been used, the Pint/PDFBox combination gives us full control over how the PDF is generated. In the future, this will allow us to not only encode the source document data, but also all the formatting decisions that are made in the layout optimization process, resulting in an “Editable PDF” file that can be edited in a non-destructive way [4].

Pint also takes an XML file as input, and a PDF can be generated from scratch by passing an abstract layout description with marked-up content. An example of the file format is given in Figure 2.

```
<?xml version="1.0" encoding="UTF-8"?>

<doc-spec>

    <!-- external fonts -->
    <font name="libertine" file="LinLibertine_R.ttf"/>
    <font name="libertine-bold" file="LinLibertine_RB.ttf"/>
    <font name="libertine-italic" file="LinLibertine_RI.ttf"/>
    <font name="libertine-bold-italic" file="LinLibertine_RBI.ttf"/>
    <font-family name="libertine" regular="libertine" italic="libertine-italic" bold="libertine-bold" bold-italic="libertine-bold-italic"/>

    <!-- character styles -->
    <character tag="emph" font-style="italic"/>
    <character tag="b" font-weight="bold"/>
    <character tag="i" font-style="italic"/>
    <character tag="tt" font="courier"/>

    <!-- block styles -->
    <block tag="p" font="libertine" font-weight="normal" font-style="regular" font-size="8.9" alignment="justify"/>
    <block tag="li" font="libertine" font-weight="normal" font-style="regular" font-size="8.9" alignment="justify"/>

    <!-- title -->
    <block tag="h1" font="biolinum" font-weight="bold" font-style="regular" font-size="17.5" alignment="center"/>

    <!-- headings -->
    <block tag="h2" font="libertine" font-weight="bold" font-style="regular" font-size="11" alignment="left"/>
    <block tag="h3" font="libertine" font-weight="bold" font-style="regular" font-size="11" alignment="left"/>

    <!-- inter-block styles -->
    <spacing from="h1" amount="9"/>
    <spacing to="h1" amount="6"/>
    <spacing to="h2" amount="8"/>
    <spacing from="h2" amount="3"/>
    <spacing to="h3" amount="8"/>
    <spacing from="h2" to="h3" amount="3"/>
    <spacing from="h3" amount="3"/>

    <!-- page layout specification-->
    <page size="letter" double="false" left-margin="54" right-margin="54" top-margin="82" top-margin-first="65" bottom-margin="82">

        <!-- Title -->
        <h1>Title</h1>

        <multi-col num-cols="2">

            <h2>ABSTRACT</h2>
            <p>
                <!-- abstract text goes here -->
            </p>

            <!-- main content begins here -->
            <h2>1<hspace/>INTRODUCTION</h2>

            <p>
                For better or for worse, PDF is the standard for exchanging scholarly articles. As the "lowest common denominator",
                it accurately preserves the article's content and presentation, regardless of whether a Word-, LaTeX- or
```

Figure 2. Sample of input file to the Pint formatter

In order to generate the input file, a two-step process is used: First, an XSLT transformation is used to copy all the content from the file and insert the stylesheet information. Afterwards, the file is programmatically read and references are sorted and relabelled before being passed to the Pint formatter. After creation of the PDF, the JATS-XML source content is attached to the PDF file and further optimization is performed.

As Pint is still in beta, some manual fine tuning was necessary to ensure that the resulting layout was attractive and, for example, the figure placement was optimal. (Note, however, that this problem also exists with other formatting engines, such as LaTeX.) We are currently working on an improved algorithm for figure placement and adjustment, with the goal of performing this task fully automatically.

3. Conclusions

This exercise has shown us that now is a good time to start thinking of moving to more structured workflows for scholarly publishing. In order to ensure take-up by the scholarly community, it is essential that the tools that are used are open-source and freely available: Both Texture and Pint are published with permissive licences (MIT and Apache respectively) and are under active development. Therefore, they can be expected to become more stable and fully-featured in the near future.

The lack of take-up of current standards is likely also due to the fact that many people are uncomfortable with the concept of Web-first content that has no authoritative visual representation. Whereas responsive document formats are more flexible and offer numerous benefits, this flexibility can, in certain cases, lead to misinterpretation of the content, which is every scientist's nightmare.

We should therefore exercise caution before moving to the "next new thing" in document formats. There is much unexplored potential in making PDF "smarter" rather than replacing it outright. This new workflow achieves the best of both worlds: as the resulting PDF files are backwardly compatible, there are fewer hurdles in its rollout; the resulting hybrid PDFs can simply be slotted into existing publishers' PDF libraries and content distribution platforms.

3.1. Alternatives

Although this pilot used two relatively young software products, the fact that they are in beta is not a reason to delay the introduction of structured workflows, as much of the technology has already been in place for a long time. For example, a good, mature alternative to Texture is LyX, which has a similar internal document model to DAR. For generation of the PDF output, pdfLaTeX can be used, and the XML metadata can be embedded afterwards using a library such as PDFBox.

The existence of these alternatives means that authors have a choice and can choose the editor that they are most familiar with; tools such as Pandoc make light work of converting between all these formats.

3.2. Next steps

The next step is to integrate structure into the reviewing process. This is a major undertaking, as it requires a much tighter integration between the reviewing platform (e.g. EasyChair), the authoring tool (e.g. Texture) and the file format.

The Texture and Pint tools are under continuous development, and work is currently progressing on improving the layout optimization algorithms to enable fully automatic generation of PDF files. It is worth noting that some of these opti-

mizations have been addressed by recent research work by the LaTeX Project [1] [2].

As the Pint/PDFBox approach gives us access to every character and layout decision made by the process, we are working with the developers to create fully tagged, accessible PDF (PDF/UA conformant), as well as embed a full layout description to make the PDF editable and fully reflowable[1]. Thus, a future reviewing platform will have all the information it needs to be able to merge the reviewers' comments and, if desired, display them on the PDF, reflowing and repaginating the document automatically.

3.3. Visual documents

Up to now, structured authoring has usually meant losing control over the visual appearance of the final document. The goal of the Editable PDF Initiative, under which the Pint formatter is being developed, is to better harmonize the document's visual presentation with its underlying source. Although the project is still at an early stage, we plan to show how structured authoring can be introduced into fields where visual communication is of higher importance and the Editable PDF format has been developed in such a way as to enable WYSIWYG editing and manipulation of the content.

But even in the sciences, it has often been questioned whether visual presentation can be fully separated from content. Authors often have the desire to maintain some control over the final layout of their works; even with rigid style guidelines set by publishers, the current LaTeX-based workflows have given authors a certain amount of flexibility to e.g. adjust figure placement and, more importantly, scrutinize the final visual result.

Bibliography

[1] Mittelbach, Frank, 2016: *A General Framework for Globally Optimized Pagination.* In *DocEng 2016: Proceedings of the 16th ACM Symposium on Document Engineering*.

[2] Mittelbach, Frank, 2017: *Effective Floating Strategies.* In *DocEng 2017: Proceedings of the 17th ACM Symposium on Document Engineering*.

[3] Peroni, Silvio; Osborne, Francesco; Di Iorio, Angelo; Nuzzolese, Andrea Giovanni; Poggi, Francesco; Vitali, Fabio; Motta, Enrico, 2017: *Research Articles in Simplified HTML: A Web-First Format for HTML-Based Scholarly Articles.* In *PeerJ Computer Science*.

[1] Note that this goes beyond simply tagging the file. Editable PDF [7] is an ongoing research project and the specification is currently under development.

[4] Hassan, Tamir, 2018: *Towards a Universally Editable Portable Document Format.* In *DocEng 2018: Proceedings of the 18th ACM Symposium on Document Engineering.*

[5] *The ACM Symposium on Document Engineering:* `https://doceng.org`

[6] *DAR (Document ARchive):* `https://github.com/substance/dar`

[7] *The Editable PDF Initiative:* `https://editablepdf.org`

[8] *Hybrid PDFs in OpenOffice:* `https://www.oooninja.com/2008/06/pdf-import-hybrid-odf-pdfs-extension-30.html`

[9] *JATS-XML:* `https://jats.nlm.nih.gov/archiving/1.1/`

[10] *Pint (Pint is not TeX):* `https://github.com/tamirhassan/pint-publisher`

[11] *Texture:* `http://substance.io/texture/`

On the Specification of Invisible XML

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Abstract

Invisible XML (ixml) is a method for treating non-XML documents as if they were XML.

After a number of design iterations, the language is ready for specification. This paper describes decisions made during the production of the specification of ixml. An interesting aspect of this specification is that ixml is itself an application of ixml: the grammar describes itself, and therefore can be used to parse itself, and thus produce an XML representation of the grammar. We discuss the decisions taken to produce the most useful XML version possible.

Keywords: XML, ixml, parsing, document formats

1. Introduction

Invisible XML (ixml) is a method for treating non-XML documents as if they were XML.

This gives a number of advantages:

- it enables authors to write documents and data in a format they prefer,
- provides XML for processes that are more effective with XML content,
- opens up documents and data that otherwise are hard to import into XML environments.

The ixml process works by providing a description of the data or document of interest in the form of a (context-free) grammar. This grammar is then used to parse the document, and the resulting parse tree is then serialised as an XML document. The extra information is included in the grammar about how the tree should be serialised, allowing the eliding of unnecessary nodes, and the choice of serialising nodes as XML elements or attributes.

As a (necessarily small) example, take the definition of the email type from XForms 2.0 [6], which is defined by a regular expression:

```
<xs:simpleType name="email">
  <xs:restriction base="xs:string">
    <xs:pattern value="([A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-~]+)
                      (\.[A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-~]+)*
```

```
                         @([A-Za-z0-9]([A-Za-z0-9-]*[A-Za-z0-9])?)
                        (\.[A-Za-z0-9]([A-Za-z0-9-]*[A-Za-z0-9])?)+"/>
    </xs:restriction>
  </xs:simpleType>
```

If we turn this into ixml, we get the following:

```
email:  user, "@", host. {An email address has two parts separated by an @
sign}
user:   atom+".".        {The user part is one or more atoms, separated by
dots}
atom:   char+.           {An atom is a string of one or more 'char'}
host:   domain+".".      {A host is a series of domains, separated by dots}
domain: word+"-".        {A domain may contain a hyphen, but not start or
end with one}
word:   letgit+.         {A domain otherwise consists of letters and digits}
-letgit: ["A"-"Z"; "a"-"z"; "0"-"9"].
-char:   letgit; ["!#$%&'*+-/=?^_`{|}~"]. {A char is a letter, digit, or
punctuation.}
```

If we now use this grammar to parse the string

```
~my_mail+{nospam}$?@sub-domain.example.info
```

and serialise the result, we get the following XML:

```
<email>
   <user>
      <atom>~my_mail+{nospam}$?</atom>
   </user>@
   <host>
      <domain>
         <word>sub</word>-
         <word>domain</word>
      </domain>.
      <domain>
         <word>example</word>
      </domain>.
      <domain>
         <word>info</word>
      </domain>
   </host>
</email>
```

If the rule for `letgit` hadn't had a dash before it, then, for instance, the element `<word>sub</word>`, would have looked like this:

```
<word><letgit>s</letgit><letgit>u</letgit><letgit>b</letgit></word>
```

Since the word part of a domain has no semantic meaning, we can exclude it from the serialisation by changing the rule for `word` into:

```
-word: letgit+.
```

to give:

```
<email>
   <user>
      <atom>~my_mail+{nospam}$?</atom>
   </user>@
   <host>
      <domain>sub-domain</domain>.
      <domain>example</domain>.
      <domain>info</domain>
   </host>
</email>
```

If we change the rule for `atom` and `domain` into

```
-atom: char+.
-domain: word+"-".
```

we get:

```
<email>
   <user>~my_mail+{nospam}$?</user>@
   <host>sub-domain.example.info</host>
</email>
```

Finally, changing the rules for `user` and `host` to:

```
@user: atom+".".
@host: domain+".".
```

gives:

```
<email user='~my_mail+{nospam}$?' host='sub-domain.example.info'>@</email>
```

To get rid of the left-over "@", we can change the rule for `email` to:

```
email: atoms, -"@", host.
```

to give:

```
<email user='~my_mail+{nospam}$?' host='sub-domain.example.info'/>
```

What we can see here is that the definition is much more structured than a regular expression, and that we have a lot of control over what emerges from the XML serialisation.

It should be noted that although this example uses a regular expression as example, ixml is more powerful than the regular expressions, being able to handle any context-free grammar.

It should also be noted that an ixml processor is not a parser generator, but a parameterised parser (although it would be possible to use the ixml format as input to a suitably general parser generator.)

Another difference with traditional parser-generator approaches, is that no lexical analysis phase is necessary, and so there is no need to define a separate class of token symbols, such as in the REx system [7].

After a number of design iterations ???, the language is now ready for specification. A draft specification [5] has been produced. This paper describes the specification, and some of the decisions made.

2. The Grammar

The grammar is based on a format known as 1VWG, a one-level variant of the two-level van Wijngaarden grammars [9].

A VWG grammar consists of a series of rules, where a rule consists of a name, a colon, a one or more 'alternatives' separated by semicolons, all terminated by a dot. An alternative consists of zero or more 'terms', separated by commas.

Expressing this in ixml looks like this:

```
ixml: rule+.
rule: name, ":", alternatives, ".".
alternatives: alternative+";".
alternative: term*",".
```

This introduces some of the extensions ixml has added to vwgs. Appending a star or plus to a term is used in many grammar systems and related notations such as regular expressions, to express repetition of a term, zero or more for a star and one or more for a plus. An ixml difference is to make both postfix operators infix as well. The right hand operand then defines a separator that comes *between* the repeated terms.

Note that these additions do not add any power to the language (they can all be represented by other means). However they do add ease of authoring, since the alternative way of writing them is verbose, and obscures the purpose of the extra rules that have to be added.

A `term` is a factor, an optional factor, or a factor repeated zero or more, or one or more times:

```
term: factor; option; repeat0; repeat1.
option: factor, "?".
repeat0: factor, "*", separator?.
repeat1: factor, "+", separator?.
separator: factor.
```

These rules also demonstrate the use of a question mark to indicate an optional factor.

A factor is a terminal, a nonterminal, or a bracketed series of alternatives:

```
factor: terminal; nonterminal; "(", alternatives, ")".
```

3. Nonterminals and Terminals

A nonterminal is just a name, which refers to the rule defining that name:

```
nonterminal: name.
```

Terminals are the elements that actually match characters in the input; Unicode characters are used. There are two forms for terminals: literal strings, and character sets.

The simplest form is a literal string, as we have seen above, such as `":"` and `","`. Strings may be delimited by either double quotes or single: `":"` and `':'` are equivalent. If you want to include the delimiting quote in a string, it should be doubled: `"don't"` and `'don''t'` are equivalent.

```
terminal: literal; charset.
literal: quoted; encoded.
quoted: '"', dchar+, '"';
        "'", schar+, "'".
dchar: ~['"']; '""'.
schar: ~["'"]; "''".
```

This introduces the *exclusion*: the construct `~['"']` matches any character *except* what is enclosed in the brackets, and is defined below.

In order to express characters with no explicit visible representation or with an ambiguous representation there is a representation for encoded characters. For instance `#a0` represents a non-breaking space.

```
encoded: "#", hex+.
hex: ["0"-"9"; "a"-"f"; "A"-"F"].
```

Encoded characters do not appear within strings, but are free-standing; however, this doesn't restrict expressiveness, since a rule like

```
end: "the", #a0, "end".
```

represents the seven characters with a non-breaking space in the middle.

Character sets, which we have also seen in the earlier simple example, allow you to match a character from a set of characters, such as `["A"-"Z"; "a"-"z"; "0"-"9"]`. Elements of a character set can be a literal string, representing all the characters in the string, a range like above, or a character class. Unicode defines a number of classes, such as *lower case letter* and *upper case letter*, encoded with two-letter abbreviations, such as *Ll* and *Lu*; these abbreviations may be used to save the work of having to define those classes yourself:

```
charset:   inclusion; exclusion.
inclusion: "[", member+, "]".
exclusion: "~", inclusion.
member:   quoted; range; class.
```

```
range:  char, "-", char.
class:  letter, letter.
char:   '"', dchar, '"';
        "'", schar, "'";
        encoded.
letter: ["a"-"z"; "A"-"Z"].
```

4. What's in a Name?

The question arose: what should be allowed as a name in ixml?

In the XML specification [8], the characters that are permitted in a name are specified almost entirely in terms of ranges over hexadecimal numbers, without further explanation. In an informal test of a handful of XML users, we found they were unable to confidently determine if certain element names were valid XML or not. Just to give you a taste of the problem, this character may not appear in an XML identifier: µ, while this one may: μ. They are both versions of the Greek letter *mu*, but one is at #B5 and the other at #3BC.

So should ixml use the same definition, or is it possible to create something more transparent? Does ixml even need to exactly duplicate the XML rules? Not all ixml names are serialised, so there is no *a priori* need for them to adhere to XML rules; furthermore, there may in the future be other serialisations; it could be left to the author to make sure that names adhered to the rules needed for the chosen serialisation. Would there be holes, and would it matter?

So what does XML allow? This is the XML rule:

```
NameStartChar   ::=
                ":" |
                [A-Z] |
                "_" |
                [a-z] |
                [#xC0-#xD6] |
                [#xD8-#xF6] |
                [#xF8-#x2FF] |
                [#x370-#x37D] |
                [#x37F-#x1FFF] |
                [#x200C-#x200D] |
                [#x2070-#x218F] |
                [#x2C00-#x2FEF] |
                [#x3001-#xD7FF] |
                [#xF900-#xFDCF] |
                [#xFDF0-#xFFFD] |
                [#x10000-#xEFFFF]
NameChar   ::=
                NameStartChar |
                "-" |
```

```
 "." |
[0-9] |
#xB7 |
[#x0300-#x036F] |
[#x203F-#x2040]
```

So an XML name can start with a letter, a colon, an underscore, and "other stuff"; it can continue with the same characters, a hyphen, a dot, the digits, and some more other stuff.

In brief, the "other stuff" for a start character is anything between #C0 (which is À) and #EFFFF (which is not assigned) -- mostly representing a huge collection of letters from many languages. What is *excluded* is:

- #D7 (×, the multiplication sign)
- #F7 (÷, the division sign)
- #300-#36F (combining characters, such as the combining grave accent),
- #37E (the Greek question mark ";")
- #2000-#200B (various spaces, such as the en space)
- #200E-#2006F, (various punctuation, including several hyphen characters)
- #2190-#2BFF, (a large number of symbol-like characters, such as arrows)
- #2FF0-#3000, (ideographic description characters)
- #D800-#F8FF, (surrogates, and private use)
- #FDD0-#FDEF, (unassigned)
- #FFFE and #FFFF (unassigned).

A name continuation character consists of the same characters *plus* as already said a hyphen, a dot, and the digits, and then #B7 (the middot "·"), the combining characters left out of the start characters, and then the two characters #203F and #2040, the overtie and undertie characters ‿⁀.

On the other hand, Unicode [10] has 30 character classes:

Name	Description	Number	Examples
Cc	Control	65	Ack, Bell, Backspace, Tab, LF, CR, etc
Cf	Format	151	Soft hyphen, Arabic Number Sign, Zero-width space, left-to-right mark, invisible times, etc.
Co	Private Use		#E000-#F8FF
Cs	Surrogate		#D800-#DFFF
Ll	Lowercase Letter	2,063	a, µ, ß, à, æ, ð, ñ, π, Latin, Greek, Coptic, Cyrillic, Armenian, Georgian, Cherokee, Glagolitic, many more

Name	Description	Number	Examples
Lm	Modifier Letter	250	letter or symbol typically written next to another letter that it modifies in some way.
Lo	Other Letter	121,047	ª, º, ƻ, dental click, glottal stop, etc., and letters from languages that don't have cased letters, such as Hebrew, Arabic, Syriac, ...
Lt	Titlecase Letter	31	Mostly ligatures that have to be treated specially when starting a word.
Lu	Uppercase Letter	1,702	A, Á, etc
Mc	Spacing Mark	401	Spacing combining marks, Devengari, Bengali, etc.
Me	Enclosing Mark	13	Combining enclosing characters such as "Enclosing circle"
Mn	Nonspacing Mark	1763	Combining marks, such as combining grave accent.
Nd	Decimal Number	590	0-9, in many languages, mathematical variants,
Nl	Letter Number	236	Ⅰ, Ⅱ, Ⅲ, Ⅳ,...
No	Other Number	676	subscripts, superscripts, fractions, circled and bracketed numbers, many languages
Pc	Connector Punctuation	10	_, ‿, ⁀, ...
Pd	Dash Punctuation	24	-, ‒, —, ...
Pe	Close Punctuation	73	),], }, ...
Pf	Final Punctuation	10	», ’, ”, ...
Pi	Initial Punctuation	12	«, ‘, “, ...
Po	Other Punctuation	566	!, @, #, ", #, %, &, ', *, ,, ., /, :, ;, ?, ¶, ...
Ps	Open Punctuation	75	(, [, {, ...
Sc	Currency Symbol	54	$, £, €, ¢, ¥, ¤, ...
Sk	Modifier Symbol	121	^, ´, `, ¨, ˚, ...
Sm	Math Symbol	948	+, <, =, >, |, ~, ±, ×, ÷, ...
So	Other Symbol	5855	©, ®, °, various arrows, much more.

Name	Description	Number	Examples
Zl	Line Separator	1	But not cr, lf.
Zp	Paragraph Separator	1	
Zs	Space Separator	17	space, nbsp, en quad, em quad, thin space, etc. Not tab, cr, lf etc.

The final decision was made to define ixml names using Unicode character classes, while keeping as close as possible to the spirit of what is allowed in XML:

```
name: namestart, namefollower*.
namestart: ["_"; Ll; Lu; Lm; Lt; Lo].
namefollower: namestart; ["-.·‿⁀"; Nd; Mn].
```

Consequently there are small differences in what a name is. For instance, Unicode classes the characters ª and º as letters, and does class both *mu* characters as letters, while XML doesn't; as was indicated above, not all ixml names are serialised, so it is the responsibility of the ixml author to ensure that those that do, adhere to the XML rules.

5. Spaces and comments

One thing that the above grammar *hasn't* yet defined is where spaces may go.

These are allowed after any token, and before the very first token, so we have to update all the rules to indicate this. For instance:

```
ixml: S, rule+.
rule: name, S, ":", S, alternatives, ".", S.
alternatives: alternative+(";", S).
```

where `S` defines what a space is:

```
S: (whitespace; comment)*.
whitespace: [Zs; #9 {tab}; #a {lf}; #d {cr}].
comment: "{", (cchar; comment)*, "}".
cchar: ~["{}"].
```

Here we see the use of a character class, `Zs`, which are all characters classified in Unicode as a space character; to this we have added tab, line feed, and carriage return (which are classified as control characters in Unicode).

Actually, Unicode is somewhat equivocal about whitespace. Along with the character class Zs it also has the character property WS ('WhiteSpace').

The Zs class contains the following 17 characters:

space, no-break space, ogham space mark, en quad, em quad, en space, em space, three-per-em space, four-per-em space, six-per-em space, figure space, punctuation

space, thin space, hair space, narrow no-break space, medium mathematical space, ideographic space.

These all have the WS property, with the exception of no-break space, and narrow no-break space, which have the property CS (common separator, along with commas, colons, slashes and the like). There are also two characters that have the WS property, but are not in the Zs class, form-feed, which has class Cc (control character), and line separator, which is the sole member of class Zl.

For the record, line feed and carriage return are both in character class Cc (control character), with property 'Paragraph Separator'; the other characters that share this property are: #1C, #1D, #1E ('information separator' control characters, all in class Cc), #85 (Next line, also in Cc), and #2029 (paragraph separator, the sole member of Zp, paragraph separator). The tab character, also in Cc, has property 'Segment Separator'; other characters that have this property are #B (line tabulation), and #1F ('Information separator), both in class Cc.

Wherever whitespace is permitted in ixml, so is a comment. A comment may itself contain a comment, which enables commenting out sections of a grammar.

An addition that has also been made is to allow `=` and well as `:` to delimit a rule name, and `|` as well as `;` to delimit alternatives:

```
rule: name, S, ["=:"], S, alternatives, ".", S.
alternatives: alternative+([";|"], S).
```

6. Serialisation and Marks

Once a grammar has been defined, it is then used to parse input documents; a resulting parse is serialised as XML. It is not specified which parse algorithm should be used, as long as it accepts all context-free grammars, and produces at least one parse of any document that matches the grammar.

By default, a parse tree is serialised as XML elements: the parse tree is traversed in document order (depth first, left to right), and nonterminal nodes are output as XML elements, and terminals are just output.

For instance, for this small grammar for simple expressions:

```
expr: operand+operator.
operand: id; number.
id: letter+.
number: digit+.
letter: ["a"-"z"].
digit: ["0"-"9"].
operator: ["+-×÷"].
```

parsing the following string:

```
pi×10
```

would produce

```
<expr>
   <operand>
      <id>
         <letter>p</letter>
         <letter>i</letter>
      </id>
   </operand>
   <operator>×</operator>
   <operand>
      <number>
         <digit>1</digit>
         <digit>0</digit>
      </number>
   </operand>
</expr>
```

To control serialisation, marks are added to grammars.

There are three options for serialising a nonterminal node, such as `expr`, and `operand` above:

1. Full serialisation as an element, as above, which is the default;
2. As an attribute: in which case all (serialised) terminal descendents of the node become the value of the attribute;
3. Partial serialisation, only serialising the children, essentially the same as option 1, but without the surrounding tag.

For serialising a terminal the only option is between serialising it and not.

There are two places where a nonterminal can be marked for serialising: at the definition of the rule for that nonterminal, which specifies the default way it is serialised, or at the use of the nonterminal, which overrides the default. A terminal can only be marked at the place of use.

There are three type of mark: "^" for full, "@" for attribute (which doesn't apply to terminals), and "-" for partial (which causes terminals not to be serialised).

To support this, we add to the grammar for `rule`, and `nonterminal`:

```
rule: (mark, S)?, name, S, ":", S, alternatives, ".", S.
nonterminal: (mark, S)?, name, S.
mark: ["@^-"].
```

and similar for terminals.

7. Attribute lifting

The only unusual case for serialisation is for attribute children of a partially serialised node. In that case there is no element for the attributes to be put on, and so they are lifted to a higher node in the parse tree. For instance, with:

```
expr: operand+operator.
operand: id; number.
id: @name.
name: letter+.
number: @value.
value: digit+.
letter: ["a"-"z"].
digit: ["0"-"9"].
operator: ["+-×÷"].
```

the default serialisation for `pi×10` would look like this:

```
<expr>
   <operand>
      <id name='pi'/>
   </operand>
   <operator>×</operator>
   <operand>
      <number value='10'/>
   </operand>
</expr>
```

However, if we changed the rules for `id` and `number` to partial serialisation:

```
-id: @name.
-number: @value.
```

so that neither produces an element, then the attributes are moved up, giving:

```
<expr>
   <operand name='pi'/>
   <operator>×</operator>
   <operand value='10'/>
</expr>
```

8. Ambiguity

A grammar may be ambiguous, so that a given input may have more than one possible parse.

For instance, here is an ambiguous grammar for expressions:

```
expr: id; number; expr, operator, expr.
id: ["a"-"z"]+.
number: ["0"-"9"]+.
operator: "+"; "-"; "×"; "÷".
```

Given the string `a÷b÷c`, this could produce either of the following serialisations:

```
<expr>
   <expr>
      <id>a</id>
```

```
    </expr>
    <operator>÷</operator>
    <expr>
      <expr>
        <id>b</id>
      </expr>
      <operator>÷</operator>
      <expr>
        <id>c</id>
      </expr>
    </expr>
  </expr>
```

or

```
  <expr>
    <expr>
      <expr>
        <id>a</id>
      </expr>
      <operator>÷</operator>
      <expr>
        <id>b</id>
      </expr>
      <id>a</id>
    </expr>
    <operator>÷</operator>
    <expr>
      <id>c</id>
    </expr>
  </expr>
```

i.e. it could be interpreted as `a÷(b÷c)` or as `(a÷b)÷c`.

In the case of ambiguous parses, one of the parse trees is serialised (it is not specified which), but the root element is marked to indicate that it is ambiguous.

Other examples of possible ambiguity to look out for are if we had defined a `rule` in the grammar as:

```
rule: name, ":", alternatives, ".".
alternatives: alternative*";".
alternative: term*",".
```

then an empty rule such as:

```
empty: .
```

could be interpreted equally well as a rule with no alternatives, or with one alternative with no terms.

Similarly, if a grammar says that spaces could appear before or after tokens

```
id: S, letter+, S.
operator: S, ["+-×÷"], S.
```

then with an input such as `a + b` the first space could be interpreted as either following `a`, or preceding `+`.

As a side note, this is why commas are needed between terms in an ixml alternative. Otherwise you wouldn't be able to see the difference between:

```
a: b+c.
```

and

```
a: b+, c.
```

9. The ixml Serialisation

The ixml grammar is itself an application of ixml, since it is defined in its own format. That means that the grammar can be parsed with itself, and then serialised to XML. This has consequences for the design of the grammar.

The main choice was whether to use attributes at all, and if so where. The decision taken was to put all semantic terminals (such as names) in attributes, and otherwise to use elements.

As pointed out above, spaces were carefully placed to prevent ambiguous parses, but also placed in the grammar so that they didn't occur in attribute values.

So as an example, the serialisation for the rule for `rule`, is:

```
<rule name='rule'>:
   <alt>
      <option>
         <nonterminal name='mark'/>?</option>,
      <nonterminal name='name'/>,
      <nonterminal name='S'/>,
      <inclusion>[
         <literal dstring='=:'/>]</inclusion>,
      <nonterminal name='S'/>,
      <nonterminal mark='-' name='alts'/>,
      <literal dstring='.'/>,
      <nonterminal name='S'/>
   </alt>.</rule>
```

Although all terminal symbols are preserved in the serialisation, the only ones of import are in attribute values.

Since formally it is the XML serialisation that is used as input to the parser, and text nodes in the serialisation are ignored, it is only the serialisation that matters for the parser. This means that a different ixml grammar may be used, as long as the serialisation is the same. So if the grammar looks like this:

```
<expr> ::= <id> | <number> | <expr>, <operator>, <expr>.
```

that is fine, as long as it produces the same serialisation structure.

10. Implementation

A pilot implementation of an ixml processor has been created (and used for the examples in this paper). The next step is to turn this into a full-strength implementation.

11. Future work

If you look at an ixml grammar in the right way, you can also see it as a type of schema for an XML format. Future work will look at the possibilities of using ixml to define XML formats. For instance, if we take the following XSL definition of the `bind` element in XForms 1.1:

```
<element name="bind">
 <complexType>
  <sequence minOccurs="0" maxOccurs="unbounded">
     <element ref="xforms:bind"/>
  </sequence>
  <attributeGroup ref="xforms:Common.Attributes"/>
  <attribute name="nodeset" type="xforms:XPathExpression" use="optional"/>
  <attribute name="calculate" type="xforms:XPathExpression" use="optional"/>
  <attribute name="type" type="QName" use="optional"/>
  <attribute name="required" type="xforms:XPathExpression" use="optional"/>
  <attribute name="constraint" type="xforms:XPathExpression"
use="optional"/>
  <attribute name="relevant" type="xforms:XPathExpression" use="optional"/>
  <attribute name="readonly" type="xforms:XPathExpression" use="optional"/>
  <attribute name="p3ptype" type="xsd:string" use="optional"/>
 </complexType>
</element>
```

you could express this in ixml as follows:

```
   bind: -Common, @nodeset?, -MIP*, bind*.
     MIP: @calculate; @type; @required; @constraint;
          @relevant; @readonly; @p3ptype.
     nodeset: xpath.
     calculate: xpath.
     type: QName.
     constraint: xpath.
     relevant: xpath.
     readonly: xpath.
     p3ptype: string.
```

The main hurdle is that a rule name must be unique in a grammar, and in XML attributes and elements with the same name may have different content models For instance, there is also a `bind` attribute on other elements in XForms.

Another example is the `input` element in XForms:

```
<element name="input">
   <complexType>
      <sequence>
         <element ref="xforms:label"/>
         <group ref="xforms:UI.Common" minOccurs="0"
                maxOccurs="unbounded"/>
      </sequence>
      <attributeGroup ref="xforms:Common.Attributes"/>
      <attributeGroup ref="xforms:Single.Node.Binding.Attributes"/>
      <attribute name="inputmode" type="xsd:string" use="optional"/>
      <attributeGroup ref="xforms:UI.Common.Attrs"/>
      <attribute name="incremental" type="xsd:boolean" use="optional"
                 default="false"/>
   </complexType>
</element>

<attributeGroup name="Single.Node.Binding.Attributes">
   <attribute name="model" type="xsd:IDREF" use="optional"/>
   <attribute name="ref" type="xforms:XPathExpression" use="optional"/>
   <attribute name="bind" type="xsd:IDREF" use="optional"/>
</attributeGroup>
```

which becomes:

```
input: -Common, -UICommonAtts, -Binding?, @inputmode?, @incremental?,
label, UICommon*.
Binding: (@model?, @ref; @ref, @model); @bind.
model: IDREF.
bind: IDREF.
ref: xpath.
```

If you had such definitions, you could then even design 'compact' versions of XML formats, eg for XForms:

```
bind //@open type boolean
input age "How old are you?"
```

by altering the rules above to

```
bind: -"bind" -Common, @ref?, -MIP*, bind*.
MIP: @nodeset; @calculate; @type; @required; @constraint; @relevant;
@readonly; @p3ptype.
nodeset: xpath.
```

```
calculate: -"calculate", xpath.
type: -"type", QName.
```

etc., and

```
input: -"input", -Common, -UICommonAtts, -Binding?, @inputmode?,
@incremental?, label, UICommon*.
```

etc.

12. Conclusion

A host of new non-XML documents are opened to the XML process pipeline by ixml. By defining ixml in ixml, it becomes the first large application of ixml.

Future work will allow designers to create formats in a compact version and an equivalent XML version in parallel.

13. References

Bibliography

[1] *Invisible XML*. Steven Pemberton. *Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10*. 2013. doi:10.4242/BalisageVol10.Pemberton01 . http://www.cwi.nl/~steven/Talks/2013/08-07-invisible-xml/invisible-xml-3.html .

[2] Steven Pemberton. *Data Just Wants to Be Format-Neutral*. *Proc. XML Prague 2016*. 2016. 109-120. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf .

[3] Steven Pemberton. *Parse Earley, Parse Often: How to Parse Anything to XML*. *Proc. XML London 2016*. 2016. 120-126. http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120 .

[4] Steven Pemberton. *On the Descriptions of Data: The Usability of Notations*. *Proc. XML Prague 2018*. 2018. 143-159. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf .

[5] Steven Pemberton. *Invisible XML Specification (Draft)*. CWI. 2018. https://www.cwi.nl/~steven/ixml/ixml-specification.html .

[6] Erik Bruchez et al. (eds). *XForms 2.0*. W3C. 2019. https://www.w3.org/community/xformsusers/wiki/XForms_2.0 .

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

[8] Tim Bray et al. (eds). *XML 1.0 5th edition*. W3C. 2008. https://www.w3.org/TR/2008/REC-xml-20081126/ .

[9] A. van Wijngaarden. 1974. *The Generative Power of Two-Level Grammars*. J. Loeckx. *Automata, Languages and Programming, ICALP 1974. Lecture Notes in Computer Science, vol 14*. Springer. https://doi.org/10.1007/978-3-662-21545-6_1 .

[10] Unicode Consortium. *The Unicode Standard, Chapter 4: Character Properties*. Unicode, Inc.. June 2018. https://www.unicode.org/versions/Unicode11.0.0/ch03.pdf#G2212 .

Jiří Kosek (ed.)

XML Prague 2019
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 2019

ISBN 978-80-906259-6-9 (pdf)
ISBN 978-80-906259-7-6 (ePub)