XML Prague 2022

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

June 9–11, 2022

XML Prague 2022 – Conference Proceedings

ISBN 978-80-907787-0-2 (pdf)
ISBN 978-80-907787-1-9 (ePub)

Table of Contents

General Information

Date

June 9th, 10th and 11th, 2022

Location

Prague University of Economics and Business
W. Churchill Sq. 4, 130 67 Prague 3, Czech Republic

Organizing Committee

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

Program Committee

Petr Cimprich, *Wunderman Thompson*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *Prague University of Economics and Business*
Ari Nordström, *Creative Words*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Bloomsbury Publishing plc*
Felix Sasaki, *SAP SW*
John Snelson, *MarkLogic*
Eric van der Vlist, *Dyomedea*
Priscilla Walmsley, *Datypic*
Norman Tovey-Walsh, *Saxonica*
Mohamed Zergaoui, *Innovimax*

Produced By

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

Sponsors

oXygen (https://www.oxygenxml.com)
Antenna House (https://www.antennahouse.com/)
le-tex publishing services (https://www.le-tex.de/en/)
Saxonica (https://www.saxonica.com/)
Czech Association for Digital Humanities (https://www.czadh.cz)
gds (https://www.gds.eu/)

Preface

This publication contains papers presented during the XML Prague 2022 conference.

In its 16th year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup and semantic on the Web, publishing and digital books, XML technologies for Big Data and recent advances in XML technologies. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 9–11 June 2022 at the campus of University of Economics in Prague. XML Prague 2022 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, Prague University of Economics and Business.

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

The Thursday and Saturday run in an un-conference style which provides space for various XML community meetings in parallel tracks. Friday is devoted to classical single-track format and papers from these days are published in the proceeedings.

We have to skip year 2021 because it was not possible to organize in-person conferences due to Covid-19 pandemy. We have moved conference to a more pandemic-friendly season of a year and coordinated this effort with Markup UK. Starting this year, both Markup UK and XML Prague will be held in alternate years. We are looking forward to meeting you in May/June 2023 in London and in June 2024 in Prague.

We hope that you enjoy XML Prague 2022!

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

A case study of committee-based semantic model development of XSD and JSON schemas

G. Ken Holman
<gkholman@CraneSoftwrights.com>

Keywords: Semantics, Semantic components, XML, XSD, JSON, JSON schema, Business documents, UN/CEFACT CCTS, OASIS UBL

1. Introduction

It is common for one to jump into schema design for describing the constraints on syntax streams, leveraging the flexible and powerful schema semantics to tailor the result to meet the needs. For projects working with XML syntax the common schema constraint expressions are governed by DTD, RELAX-NG, or XSD standards. For those working with JSON syntax there are a number of schema constraint expressions including JSON Schema.

Schema semantics focus on the model of the syntax, not the model of the information. Of course one leverages the schema semantics to model how the information is expressed in syntax, but that is a manual task and a bespoke result ends up being used in the project. For very large or complex semantics, bespoke schemas can be inconsistent or prone to errors, not to mention a burden on the human resources to interpret the needs for expressing the information being serialized in syntax.

On November 12, 2021 on the XML-Dev list a discussion of semantic models boiled down to a post by Hans-Jürgen Rennau in response to Michael Kay:

> *Michael: You create a semantic data model for a problem domain (for example, in UML), and then you define XML or JSON representations of the data in that domain. The conceptual data model comes first, the concrete realisations come second.*
>
> *Hans-Jürgen: This sounds to straightforward, intuitive, compelling - that it puzzles me why it seems to be done only very rarely, and concerning the approach how to do it there does not seem to be any established good practice. The OASIS work Ken points to looks exactly like a step in that direction, but probably very few people are aware of it?*

eBusiness continues to be a growing area. For 30 years (and running) an ISO committee has been standardizing Open-edi as an approach to modeling many aspects of static and dynamic semantics from a business perspective that regards

functional services as the distinct and separate implementation of those business concepts.

This paper is a case study of the Organization for the Advancement of Structured Information Standards (OASIS) Universal Business Language (UBL) committee following the Open-edi approach separating static semantic information design from syntactic data constraint expressions. The OASIS UBL committee is over 20 years old now. OASIS UBL ISO/IEC 19845 XML is used around the world in many business document interchange networks and environments. In UBL 2.3 business concepts govern 91 separate document types as onion-skins around a common core library of over 4000 information items.

For these 91 document types UBL standardizes a published set of static business document semantics and a published set of XSD schemas. User communities are expected to adopt subsets of the semantics according to their particular business needs. Never was it the intention of the committee that any one community implement every UBL business object. Nor was it ever the intention of the committee to model dynamic business semantics as the ways that UBL is being used are as varied as the committees that are using UBL.

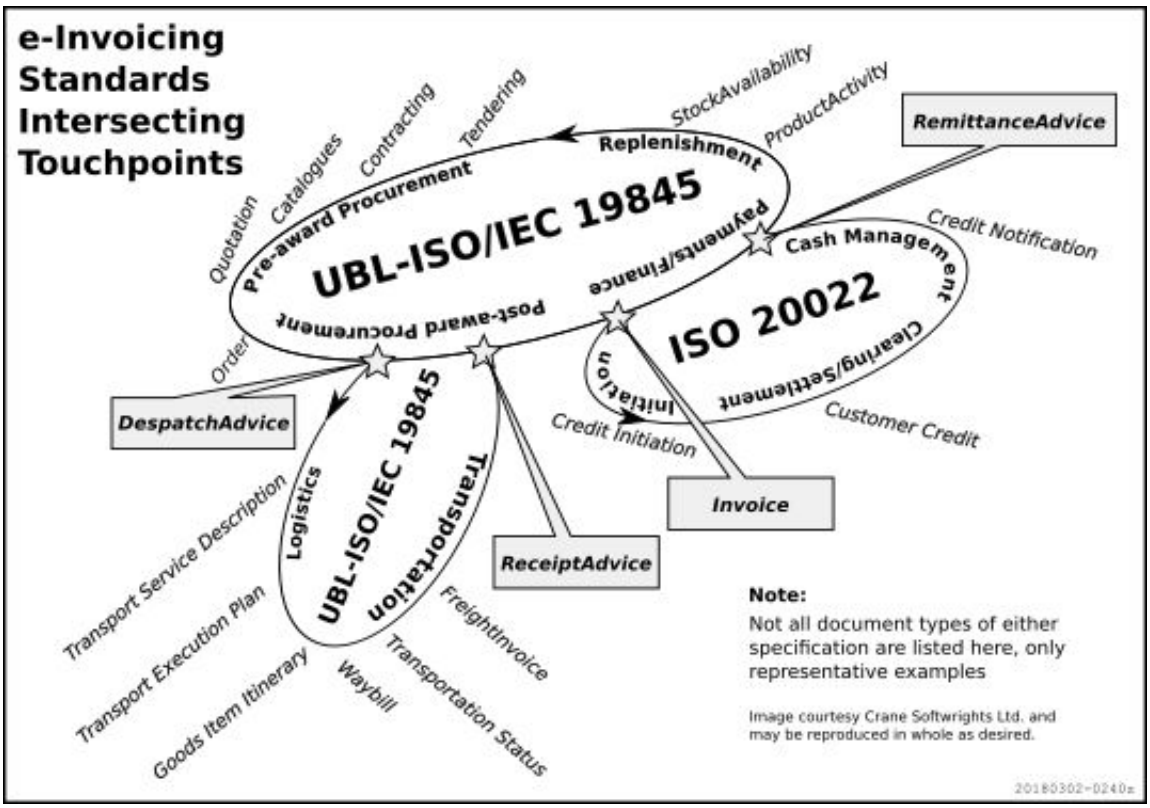


Figure 1. OASIS UBL ISO/IEC 19845

The sheer magnitude of the document specifications precludes human intervention, but such was not the reason to recognize the benefits in adopting how Open-edi separates the semantics of data from the syntax of data.

UBL was not designed using XML or XSD but, rather, the Core Component Technical Specification (CCTS) Version 2.01, a syntax-neutral modeling approach for hierarchical information found in business documents. The focus of committee members is the CCTS, whereas the XSD is machine generated without human intervention to produce validation artefacts that govern constraint checking of syntactic documents. The machine generation is governed by OASIS Business Document Naming and Design Rules (BDNDR).

The first question about having JSON schemas for the UBL document models was addressed to the committee chairman November 2012 in private correspondence. The topic first was added on a meeting agenda in August 2013. No significant discussions were held until June 2016 when the committee decided, finally, to produce some guidance on the topic before ad-hoc JSON approaches popped up in the wild. The delay was caused by a reluctance to consider JSON as a viable serialization syntax exchanged between two disparate parties, rather than consign the syntax only to serializations where one party is in control of both sides of the exchange.

Rather than directly converting the XSD to JSON, the UBL committee initiated a project to revise the BDNDR to govern the creation of JSON schemas from CCTS in addition to creating the XSD schemas from CCTS. With the release of UBL 2.3, the CCTS and XSD models are part of the official standard and the JSON Schema models are part of a published committee note.

2. Open-edi standards ISO/IEC 14662 and ISO/IEC 15944

The focus of UBL is on the static semantic data model of the data transfers (i.e. messages or documents), not on the dynamic semantics of the interpretation of the content (i.e. business in general or business processes). The UBL committee expressly limited their attention to how to structure the content, and not how to use the content, because there was no way the committee would conceive of all of the possible uses of UBL in the real world. Dynamic business relationships constantly change the way data is used and the expectations of the content of the data, and so the UBL committee elected solely to standardize the way the content is structured and serialized so that it could be exchanged readily and consistently. No longer would business document projects have to conceive of their own business object structures to convey commonly-understood eBusiness concepts.

This important distinction is seen in the way the international standardization community views “eBusiness”. In the early 1990s the joint ISO/IEC JTC 1/SC 32/WG 1 eBusiness standards committee working group created the ISO/IEC 14662 Open-edi Reference Model. This prescribes the separation of abstract business concepts from concrete functional implementations of those abstractions. This allows for identification, focus, and standardization in respective areas of effecting electronic business, while recognizing that the environment in which

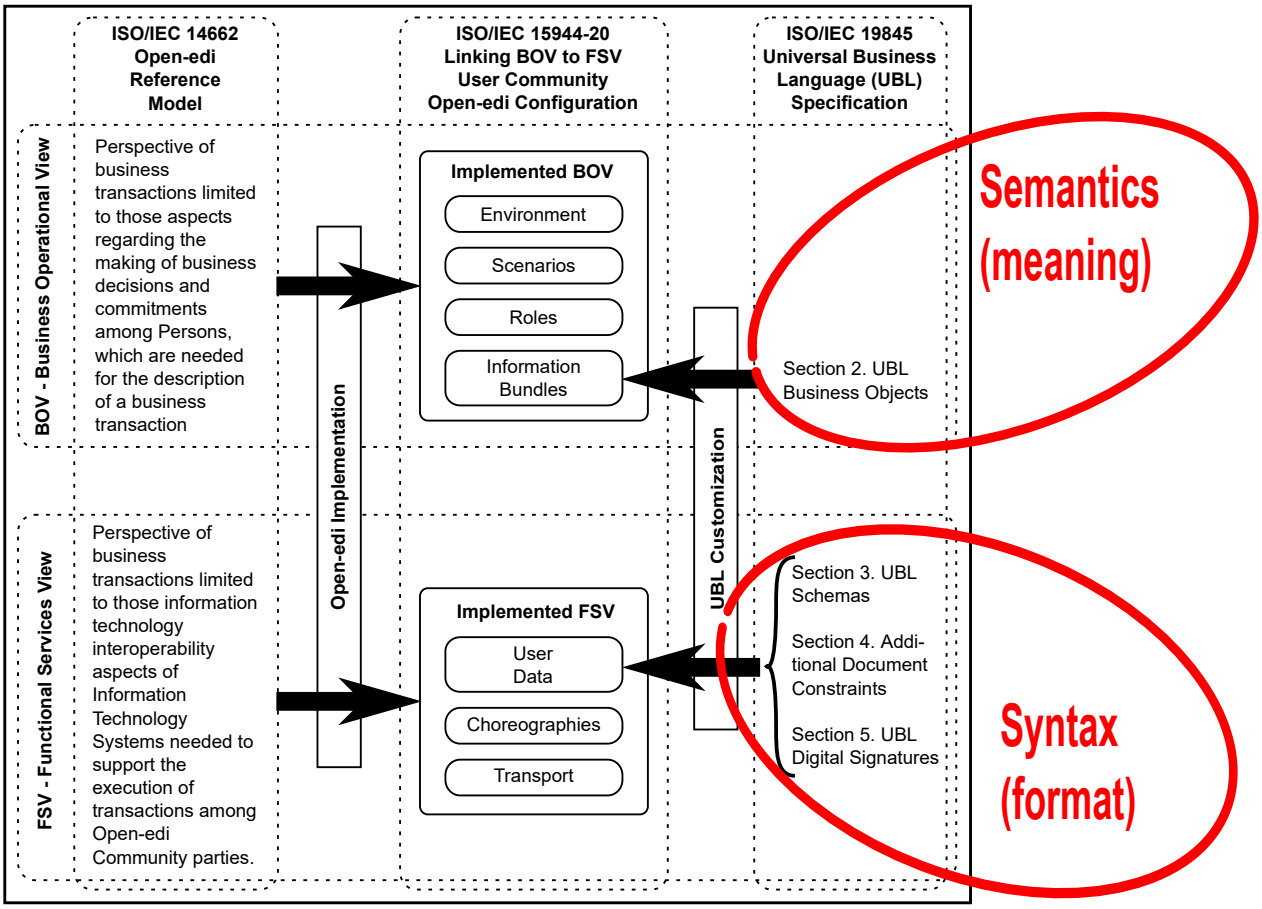


Figure 2. Open-edi standards

business operates works independently from a functional implementation of that environment, yet relies heavily on that functional implementation to be realized.

ISO/IEC 15944 Part 20 outlines how the Business Operational View (BOV) establishes the business environment in which trading partners are doing business, the specific business scenarios that are being addressed by an implementation, the various roles that are party to the information being exchanged in a given scenario, and the semantic bundles of information needed for the roles to perform their part in the trading partner scenario in the business environment. The specification also outlines how the Functional Services View (FSV) establishes the transport of content between trading partners supporting the choreography of the exchange of syntactic user data in fulfillment of the semantic bundles of information.

It is this reification of the information bundles as user data that bridges business semantics (the meaning of the data) from services implementing the semantics (the syntax of the data). The UBL specification document itself directly reflects the separation of the information bundles from the user data in the table of contents and the section content.

Also, this underscores the committee's focus only on what information is described and how it is serialized, without any focus on how the information is used: any dynamic semantics reflecting how business is performed using the information bundles is out of scope of the UBL committee and project. The only semantics being defined are those of the information bundles being exchanged.

This has contributed to the worldwide success of deploying UBL in different business environments. While the UBL committee members have created a repertoire of business objects based on general accounting and business principles, UBL user communities have cherry-picked their own set of information bundles from this. For example, the suite of UBL business objects in the information bundles used in the US Business Payments Coalition project differs slightly from the suite of objects in the bundles used in the European Peppol project.

3. CCTS: semantic modeling for business documents

In 1999 the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) worked with the Organization for the Advancement of Structured Information Standards (OASIS) to create ebXML "Electronic Business Using Extensible Markup Language" to provide an "open, XML-based infrastructure that enables the global use of electronic business information in an interoperable, secure, and consistent manner by all trading partners".

- ISO 15000-1: ebXML Collaborative Partner Profile Agreement (ebCPP)
- ISO 15000-2: ebXML Messaging Service Specification (ebMS)
- ISO 15000-3: ebXML Registry Information Model (ebRIM)
- ISO 15000-4: ebXML Registry Services Specification (ebRS)
- ISO 15000-5: ebXML Core Components Specification (CCS)

The precursor to 15000-5 is the UN/CEFACT Core Component Technical Specification (CCTS) version 2.01, which was in play at the time that UBL began its development.

CCTS defines a core set of component types with content specifications and associated supplementary components. Whereas XSD data types are suitable for any kind of data, CCTS core component types are specifically designed for constructing information bundles for business documents.

Recall the base data types of XSD, and for convenience those of JSON:

Table 1. Base simple data types of XSD and JSON

XSD	**JSON**
string and string sub-types	string
boolean	boolean

XSD	**JSON**
base64Binary	
hexBinary	
float	
decimal, integer, and integer sub-types	
double	number
anyURI	
QName	
NOTATION	
duration, date, and time types	
	array
	object
	null

Using XSD one can compose many and varied complex types on a custom basis. Any data type can be used for the element content, any attribute can be used, and any attribute can be of any XSD type that can be selected by the XSD designer.

In contrast, using CCTS one cannot compose any custom base data types, as one is obliged to use only the Core Component Types (elements in XML), their Secondary Representation Terms (derived elements in XML), and their pre-defined properties called Supplementary Components (attributes in XML):

Table 2. CCTS Core Component Types and Supplementary Components

Core Component Type (CCT)			**CCT Supplementary Components**
Name	**Base**	**Secondary**	**Name (all are strings)**
Amount	decimal		Currency Identifier
			Currency Code List Version Identifier
Binary Object	base64 binary	Graphic, Picture, Sound, Video	Format
			MIME Code
			Encoding Code
			Character Set Code
			URI

Core Component Type (CCT)			**CCT Supplementary Components**
Name	**Base**	**Secondary**	**Name (all are strings)**
			File Name
Code	normalized string		List Identifier
			List Agency Identifier
			List Agency Name
			List Name
			List Version Identifier
			Name
			Language Identifier
			List URI
			List Scheme URI
Date Time	string	Date, Time	Format
Identifier	normalized string		Scheme Identifier
			Scheme Name
			Scheme Agency Identifier
			Scheme Agency Name
			Scheme Version Identifier
			Scheme Data URI
			Scheme URI
Indicator	string		Format
Measure	decimal		Unit Code
			Unit Code List Version Identifier
Numeric	decimal	Value, Rate, Percent	Format
Quantity	decimal		Unit Code
			Unit Code List Identifier
			Unit Code List Agency Identifier
			Unit Code List Agency Name

Core Component Type (CCT)			**CCT Supplementary Components**
Name	**Base**	**Secondary**	**Name (all are strings)**
Text	string	Name	Language Identifier
			Language Locale Identifier

Outside of the XML element content and prescribed available XML attributes implementing the Core Component Types and Supplementary Components, users of CCTS are not permitted to add any other types of elements nor any other attributes of any kind to the XML.

Users of CCTS derive unqualified data types from the Core Component Types, broken down as primary and secondary representation terms. In OASIS, the following 20 unqualified data types defined in the Business Document Naming and Design Rules (BDNDR) are available for each of the abstract business objects:

Table 3. OASIS BDNDR Unqualified Data Type Restrictions

Unqualified Data Type	**Core Component Type**	**Restriction**
Amount	Amount	Required currency identifier
Binary Object	Binary Object	Required MIME Code
Code	Code	
Date Time	Date Time	xsd:dateTime
Date	Date Time	xsd:date
Time	Date Time	xsd:time
Graphic	Binary Object	Required MIME Code
Identifier	Identifier	
Indicator	Indicator	xsd:boolean
Measure	Measure	Required Unit Code
Name	Text	
Numeric	Numeric	
Percent	Numeric	
Picture	Binary Object	Required MIME Code
Quantity	Quantity	
Rate	Numeric	

Unqualified Data Type	**Core Component Type**	**Restriction**
Sound	Binary Object	Required MIME Code
Text	Text	
Value	Numeric	
Video	Binary Object	Required MIME Code

One builds hierarchical business document structures from the CCTS Core Component Types by creating three kinds of Business Information Entities (BIE). As all tree-like document hierarchies go, there are leaves with content, branches with leaves, branches with branches, and a trunk with branches.

Starting with the leaves of the tree, Basic Business Information Entities (BBIE) contain the actual document data sequences of octets, lexically constrained and structured in elements with attributes according to the unqualified data types. No octets of business content in the data stream are allowed to be anywhere other than within BBIEs.

The branches of the tree are the Associated Business Information Entities (ASBIE), each one's shape defined by a particular Library Aggregate Business Information Entity (Library ABIE). The ABIE shape contains a combination of zero or more BBIEs followed by zero or more ASBIEs. Library ABIEs are manifest as elements only as ASBIEs and not standalone on their own.

The trunks of the tree are the Document Aggregate Business Information Entities (Document ABIE) and these are the only ABIEs that are manifest directly as elements. They, too, contain a combination of zero or more BBIEs followed by zero or more ASBIEs.

UBL 2.3 has 91 document types and over 4000 constituent components.

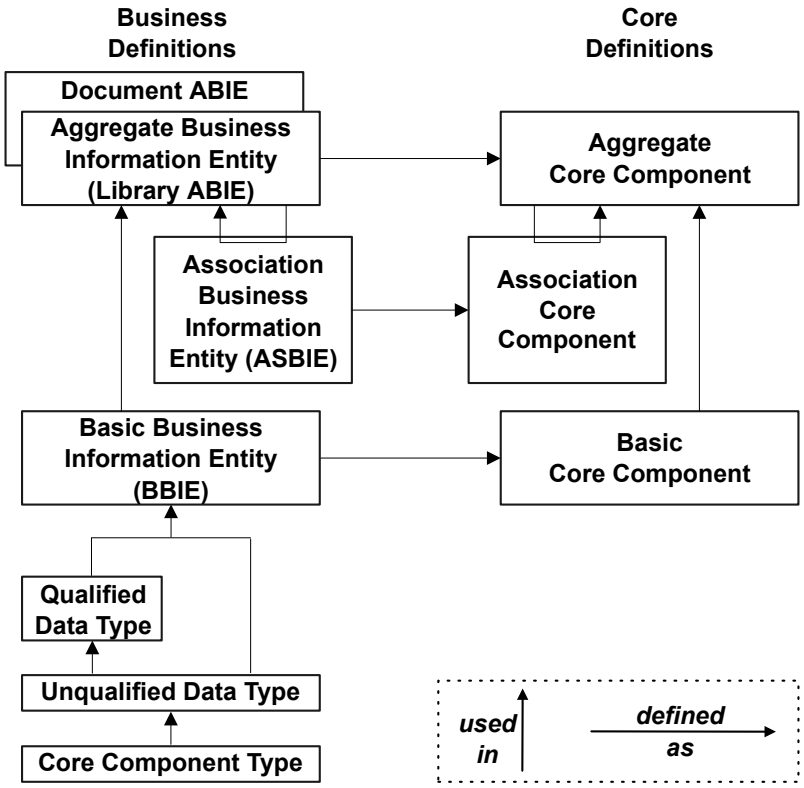


Figure 3. Business Information Elements and their defining CCTS components

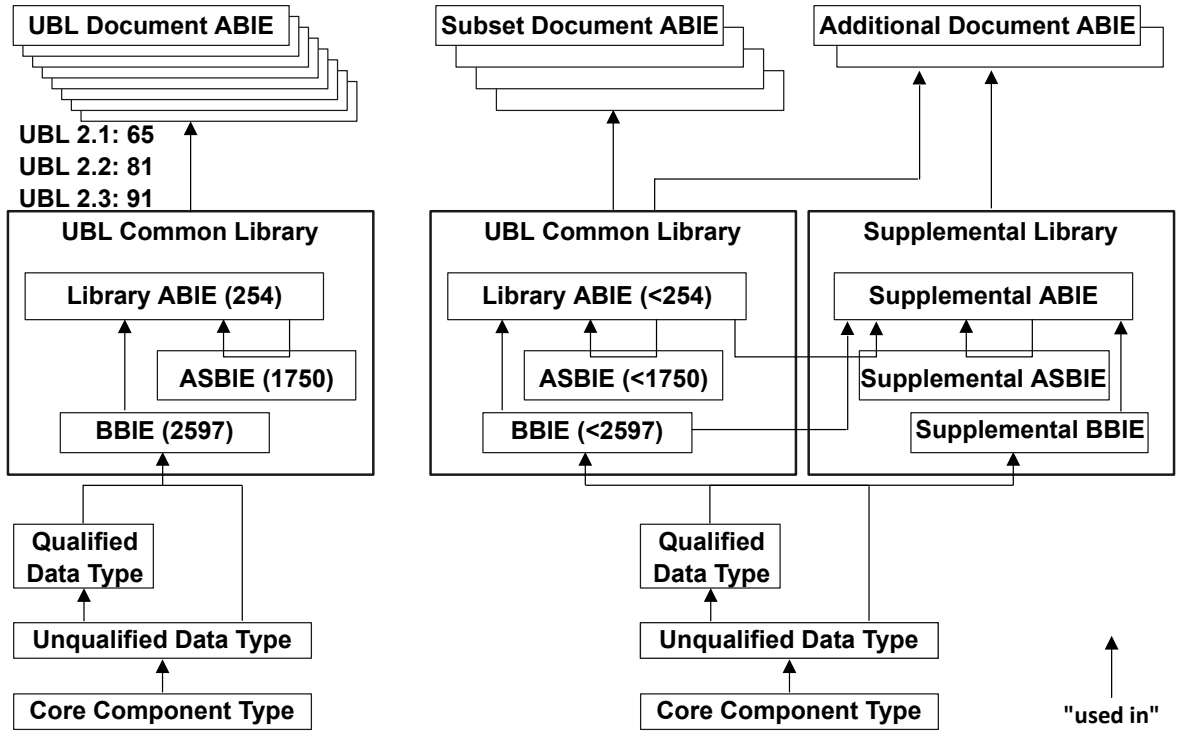


Figure 4. CCTS components in full UBL and subsets

The committee members focus on the business semantics by modeling the CCTS components for UBL in a shared Google spreadsheet. ABIEs in magenta, ASBIEs in green, BBIEs in blue. This is illustrated in the following sample semantic for a postal address:

	A	B	C	D	E
11	ActivityProperty			A class to define a name/value pair for a property of an inventory planning activity.	
12	Name		1	The name of this activity property.	
13	Value		1	The value of this activity property.	
14	Address			A class to define common information related to an address.	
15	ID		0..1	An identifier for this address within an agreed scheme of address identifiers.	DetailsKey
16	AddressTypeCode		0..1	A mutually agreed code signifying the type of this address.	
17	AddressFormatCode		0..1	A mutually agreed code signifying the format of this address.	
18	Postbox		0..1	A post office box number registered for postal delivery by a postal service provider.	PostBox, PO Bo
19	Floor		0..1	An identifiable floor of a building.	SubPremiseNu
20	Room		0..1	An identifiable room, suite, or apartment of a building.	SubPremiseNu
21	StreetName		0..1	The name of the street, road, avenue, way, etc. to which the number of the building is attached.	Thoroughfare
22	AdditionalStreetName		0..1	An additional street name used to further clarify the address.	Thoroughfare
23	BlockName		0..1	The name of the block (an area surrounded by streets and usually containing several buildings) in which this address is located.	
24	BuildingName		0..1	The name of a building.	BuildingName
25	BuildingNumber		0..1	The number of a building within the street.	PremiseNumbe
26	Description		0..n	Text describing this address for clarification or specificity	
27	InhouseMail		0..1	The specific identifable location within a building where mail is delivered.	MailStop
28	Department		0..1	The department of the addressee.	Department
29	MarkAttention		0..1	The name, expressed as text, of a person or department in an organization to whose attention incoming mail is directed; corresponds to the printed forms "for the attention of", "FAO", and ATTN:".	
30	MarkCare		0..1	The name, expressed as text, of a person or organization at this address into whose care incoming mail is entrusted; corresponds to the printed forms "care of" and "c/o".	
31	PlotIdentification		0..1	An identifier (e.g., a parcel number) for the piece of land associated with this address.	
32	CitySubdivisionName		0..1	The name of the subdivision of a city, town, or village in which this address is located, such as the name of its district or borough.	
33	CityName		0..1	The name of a city, town, or village.	LocalityName
34	PostalZone		0..1	The postal identifier for this address according to the relevant national postal service, such as a ZIP code or Post Code.	PostalCodeNun
35	CountrySubentity		0..1	The political or administrative division of a country in which this address is located, such as the name of its county, province, or state, expressed as text.	AdministrativeA Country, Shire,
36	CountrySubentityCode		0..1	The political or administrative division of a country in which this address is located, such as a county, province, or state, expressed as a code (typically nationally agreed).	AdministrativeA State Code
37	Region		0..1	The recognized geographic or economic region or group of countries in which this address is located.	LocalityName, E Zone
38	District		0..1	The district or geographical division of a country or region in which this address is located.	LocalityName, A
39	TimezoneOffset		0..1	The time zone in which this address is located (as an offset from Universal Coordinated Time (UTC)) at the time of exchange.	
40	AddressLine		0..n	An unstructured address line.	
41	Country		0..1	The country in which this address is situated.	
42	LocationCoordinate		0..n	The geographical coordinates of this address.	
43	AddressLine			A class to define an unstructured address line.	
44	Line		1	An address line expressed as unstructured text.	

Figure 5. Committee spreadsheet

The spreadsheet has no concepts of syntax, only core component data types for the BBIE basic components. The ABIE shapes are ordered by the spreadsheet with each member component's constrained cardinality. As a convention, all BBIEs of an ABIE are listed before the ASBIEs of the ABIE.

4. Business document naming and design rules

Rules that govern creating the user data schema constraint expressions from the specification of the abstract information bundles containing business objects are termed by OASIS the Naming and Design Rules (NDR). The OASIS NDR approach to describing the schema constraint expressions are different than other approaches, such as the NDR from UN/CEFACT. Most of the differences come from philosophical opinions about core component types, in particular code lists, and foreign content.

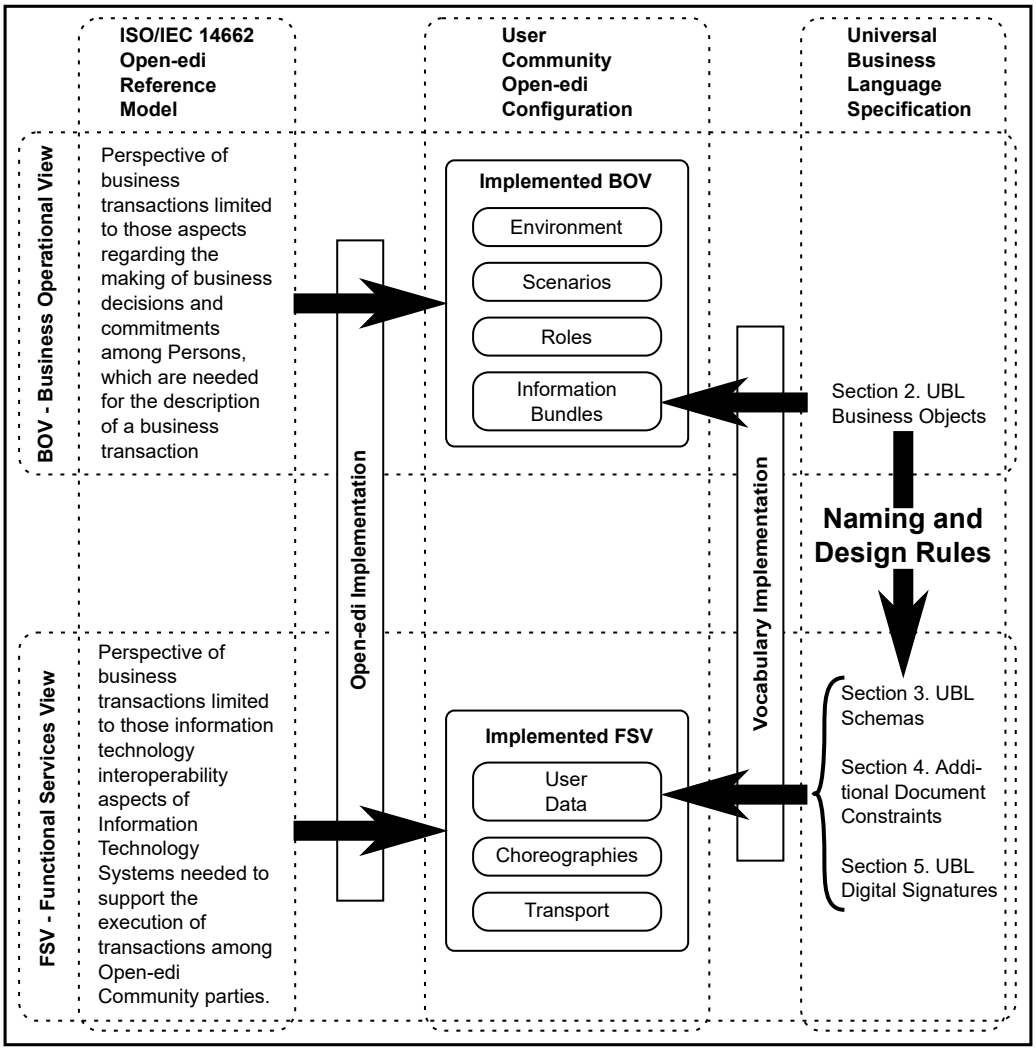


Figure 6. Role for naming and design rules

The OASIS Business Document Naming and Design Rules (BDNDR) Version 1.0 governed only the creation of XSD constraint expressions as was required at the time UBL was published.

The general user community approached the UBL committee asking for standardized structures in JavaScript Object Notation (JSON) syntax. BDNDR Version 1.1 published September 8, 2021 does not change any of the XSD NDR, it only adds JSON NDR to the specification.

Interestingly, the inherent backwards compatibility of dealing with sequences of XML elements is available in JSON only when a sequences of JSON objects is expressed in an array. This imposes the requirement that each and every BBIE and ASBIE be reified as JSON arrays. But many users of JSON balked at this constraint, preferring, rather, to express JSON objects in singularity when their maximum cardinality is one.

This bifurcates the JSON user community into those who want their legacy documents to remain JSON schema valid when future versos of the schema intro-

duce unlimited maximum cardinality for an item, and those who do not care for such.

Accordingly, there are two JSON schema expressions in the BDNDR, termed "legacy" and "model". Legacy expressions use arrays for every item in the business document so that legacy instances will validate with future schemas. Model expressions use singletons for items in the business document that have a maximum cardinality of one, sacrificing future backward compatibility for JSON instances when the a committee raises the maximum cardinality of that item.

Example XSD rules include:

FRG08 BBIE element declarations
The common BBIE schema fragment shall include an element declaration for every BBIE in the model (that is, from every Document ABIE and every Library ABIE) describing the content of each BBIE.

DCL10 BBIE element declaration

Every BBIE element shall be declared with the BBIE name as the element name and the concatenation of the BBIE name and "Type" as the type.
Example
`<xsd:element name="SourceCurrencyCode"`
`type="SourceCurrencyCodeType"/>`

Example JSON schema rules include:

FRG29 BBIE object definition declarations
The one BBIE schema fragment shall include a "definitions" object that contains an object definition declaration for every BBIE in the model (that is, from every Document ABIE and every Library ABIE). Each such declaration shall reference its corresponding qualified or unqualified data type without any title or description information.

DCL26 BBIE property declaration in an ABIE object

Legacy-mode declaration:
Every BBIE child of an ABIE shall be declared as an array named by the CCTS Component Name of the BBIE. It shall have as its title the CCTS Dictionary Entry Name. It shall have as its description the CCTS Definition. It shall have a minimum number of items as 1. If the cardinality has a maximum bound of 1, then the declaration shall have a maximum number of items as 1, otherwise there shall be no constraint on the maximum number of items. It shall declare that additional properties are not permitted. The items of the array shall be declared by referencing the BBIE declaration in the BBIE schema fragment using the CCTS Component Name of the BBIE.
Example

```
"ResponseDate": {
  "title": "Application Response. Response Date. Date",
  "description": "The date on which the information in
the response was created.",
  "items": {
    "$ref":
"../common/UBL-CommonBasicComponents-2.2.json#/
definitions/ResponseDate"
    },
  "maxItems": 1,
  "minItems": 1,
  "type": "array",
  },
```

Model-mode declaration:
All BBIE children with an unbounded maximum cardinality of 'n' are declared as arrays in the manner used for a legacy-mode declaration.
All BBIE children with a bounded maximum cardinality of "1" are declared as an object referencing the BBIE declaration in the BBIE schema fragment using the CCTS Component Name of the BBIE.
Example

```
"ResponseDate": {
  "title": "Application Response. Response Date. Date",
  "description": "The date on which the information in
the response was created.",
  "$ref":
"../common/UBL-CommonBasicComponents-2.2.json#/
definitions/ResponseDate"
  },
```

It is important to note that the OASIS BDNDR does not transliterate the XSD schemas into JSON Schema schemas. Rather, the JSON Schema schemas are

derived from the CCTS Core Component Types, the OASIS Unqualified Data Types, and the UBL Business Information Entities. The business abstractions are well enough distinct from the XSD concepts that there were no obstacles in applying JSON concepts in the reification of the abstractions.

5. Expecting the unexpected

The makeup of the original UBL technical committee included a lot of XML experience. Before the XML issues were resolved and the committee became weighted almost entirely in business experts rather than XML experts, two important distinctions developed between the UBL perspective of business documents and the UN/CEFACT perspective of business documents. Both issues relate to expecting the unexpected from our users.

The rigidity of the UN/CEFACT NDR is unpalatable to the UBL committee members. In particular, all of the code lists with sets of values in a value domain are expressed as schema enumerations, and there is no accommodation whatsoever for foreign content.

Early in the development of UBL, the committee recognized that code lists are content, not structure. And the committee wanted hands off of all content, because content is the purview of the users of UBL, not the UBL committee. Accordingly, the OASIS BDNDR does not use schema enumerations for code lists. It is expected that users will use a second pass value validation that can check code lists and many other aspects of values. How that second pass is implemented is out of the scope of UBL, but the committee has created two specifications to manage code lists: OASIS genericode for enumerating coded values and their associated metadata, and OASIS Context-value Association for mapping value checks to arbitrary hierarchical contexts.

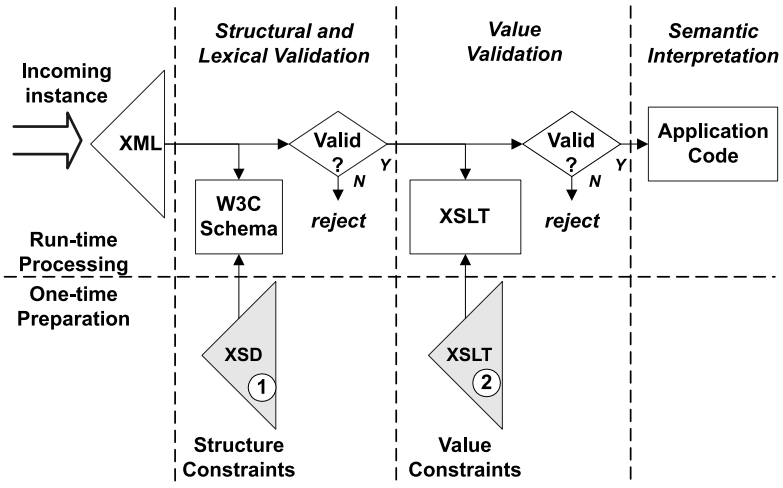


Figure 7. Two-pass validation

Also early in the development process, it was recognized that the UBL committee never will know ahead of time all of the requirements communities will have for their information bundles. As complete and as big as UBL has become, the committee expected that communities would have unexpected requirements. Yet CCTS does not accommodate foreign content, rigidly constraining users of pure CCTS models to know everything in advance.

The UBL committee addressed this in the OASIS BDNDR in a very flexible fashion through the availability of extension points. Prior to UBL 2.3, every document element at the root of the tree had an optional single extension point for foreign content. From UBL 2.3 and going forward, each and every branch of the document tree has an optional single extension point into which multiple extensions can be placed.

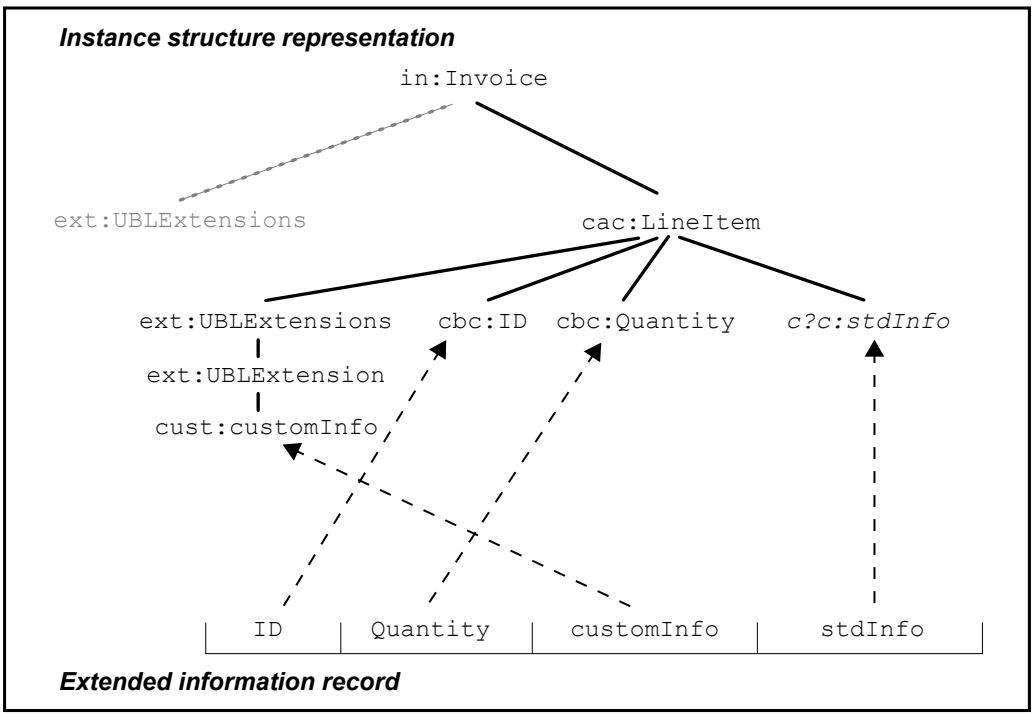


Figure 8. Extension content

User communities leveraging extensions are encouraged to use XML vocabularies that already are established and standardized by some authorities, or to re-use components from the UBL common library, but they are not prohibited from including colloquial content. The committee puts no constraints on the content of the extension point, other than encouragement to find already-standardized business objects wherever possible.

At this time, supporting foreign content extensions in the JSON serialization is not possible using the JSON schemas published by the committee. Accordingly, user communities will have to figure that out for themselves using ad-hoc methods.

6. Schema expressions

As mentioned earlier, UBL committee members focus on the online Google spreadsheets that express the document content using CCTS concepts of ABIEs, BBIEs, and ASBIEs.

The GitHub project at https://github.com/oasis-tcs/ubl has the complete sources for fetching and transforming the spreadsheet content into reports in HTML, text, ODF, and XLS formats, and into validation artefacts in OASIS Context-value Association (CVA), XSD, and JSON Schema formats according to the OASIS BDNDR Version 1.1 specification.

The act of a push check-in to GitHub triggers the complete fetch and transformation process, using OASIS genericode as a sparse table XML serialization of the CCTS semantics found in the spreadsheet:

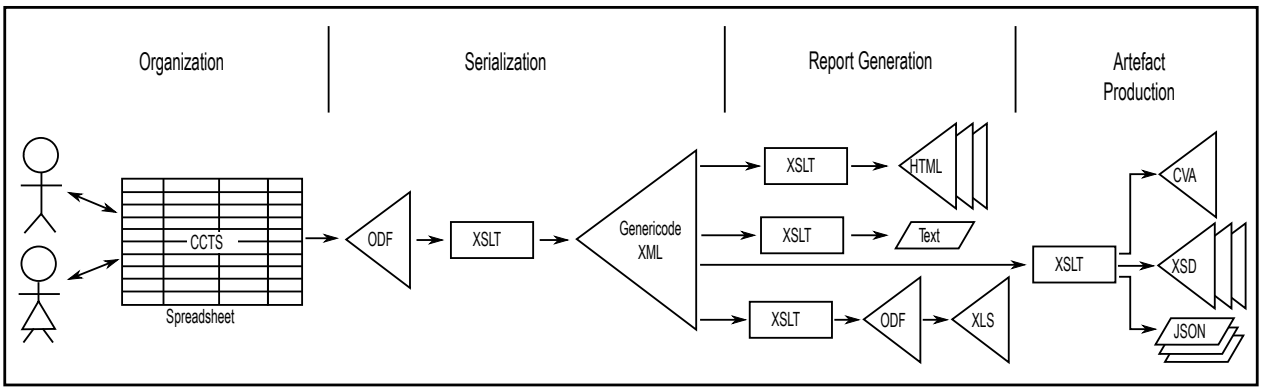


Figure 9. Artefact generation from spreadsheets

Any committee member with GitHub permissions can create their own branch, point to their own spreadsheet from that branch, and make the changes they want. The act of pushing to the repository triggers the GitHub action to produce a complete UBL deliverable package including both documentation and schema expressions.

The resulting validation artefacts correspond to the source CCTS abstractions as shown in this illustration:

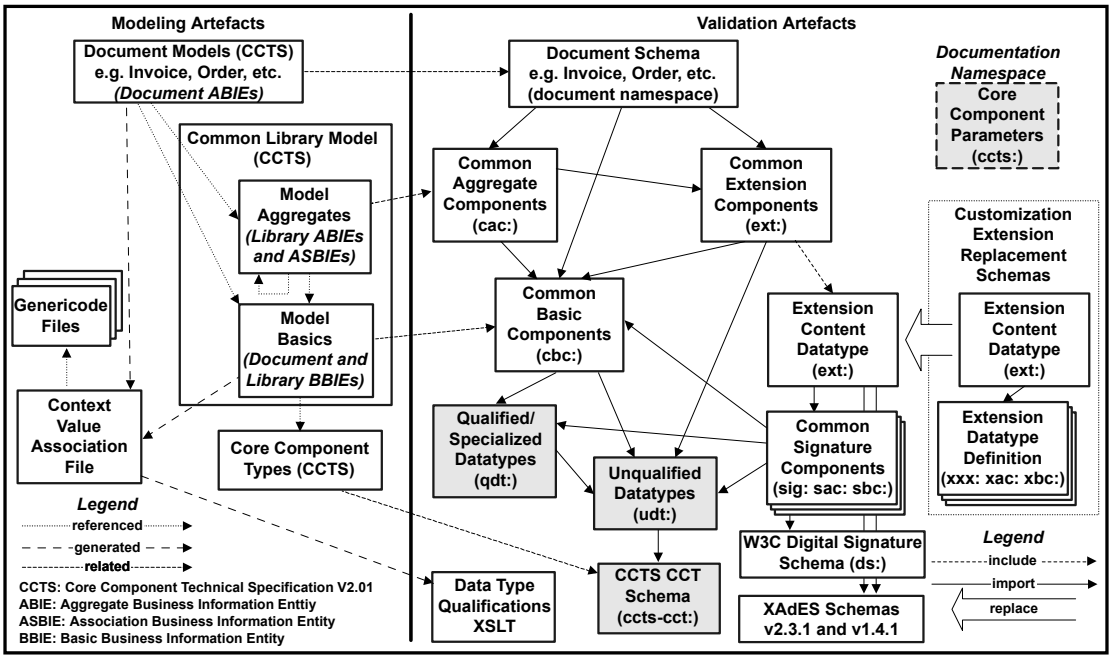


Figure 10. Schema fragment hierarchy

Note that the CCTS CCT Schema fragment indicated in the `ccts-ccts` namespace at the base of the hierarchy is the untouched schema fragment published by UN/CEFACT upon which UN/CEFACT business document schemas also are built. This promotes a base level of interoperability between the two communities, though limited to independent business objects and not to the divergent business document schemas built upon the common Core Component Types.

The apparent complexity of the tree of schema fragments is obligated by the need for namespaces to separate the different levels of constructs in the schema hierarchy. ABIEs, BBIEs, unqualified data types, and core component types all are permitted to have the same name. And user extensions may introduce names that would not to be in conflict with future versions of UBL.

7. Conclusion and implications

By building UBL document structures on CCTS instead of on XSD, UBL committee members can focus on business concepts and business-oriented data types, rather than on arbitrary XML structures. Nor need they know anything about XSD schema semantics or even XML syntax. This limitation introduces simplicity for the designers and users, and it offers consistency and predictability for programmers and downstream manipulators of the document schemas and the document content.

Separating semantics from functional implementation promotes focus on the information and not how it is expressed in syntax. Information is more important than syntax. The approach insulates decisions in semantics from decisions in syn-

tax. In doing so, any supporting syntax can be used and one is not limited to transliteration from other supported syntaxes.

The synthesis of document syntax schemas precludes the need for human intervention and the inevitable introduction of typographical errors when humans (at least me!) are involved.

The focus of semantic information bundles and their associated syntactic user data is insulated from the semantics of the world in which the information bundles are used and reified. Consider in this diagram how the European standards organization CEN created EN 16931 as semantic business profiles and profiled subsets of the UBL information bundles (perhaps inappropriately named "syntax binding", though the end result does happen to be the UBL syntax associated with the UBL business objects). The CEN profiles then are used within a legal semantic business framework defined by Peppol Business Interoperability Specifications (BIS):

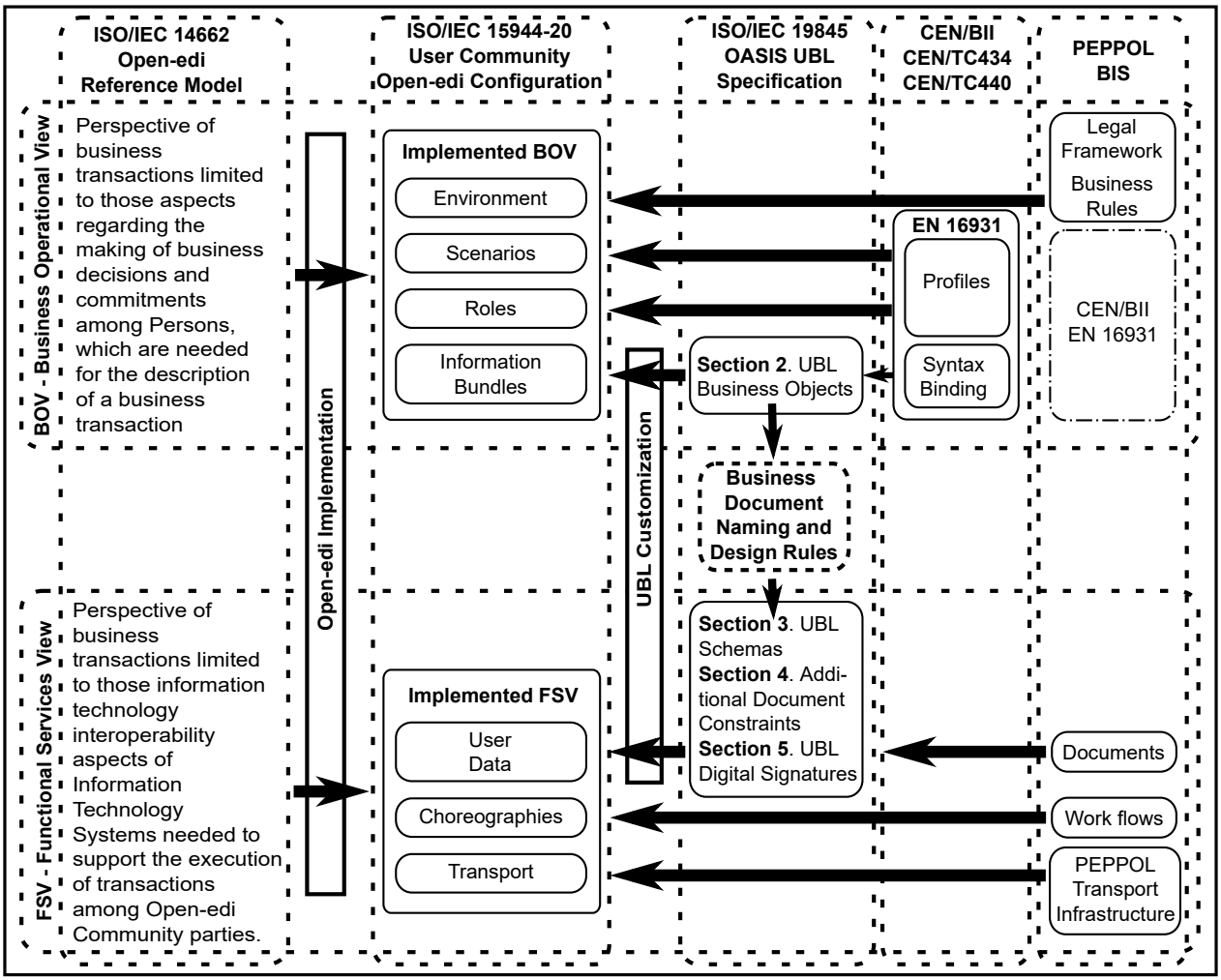


Figure 11. Application in context

What semantics govern your information set? Find an abstraction and determine the mechanical rules of producing schema expressions. Then model your document using the abstraction and mechanically generate the schema files.

And don't forget to expect the unexpected by preparing for it inevitably to arrive.

X-definition 4.2 XML, JSON, YAML, and XON

Václav Trojan
Syntea software group a.s.
<trojan@syntea.cz>

Tomáš Šmíd
Syntea software group a.s.
<smid@syntea.cz>

1. Introduction

X-definition is a language that was originally designed to work with XML data and it is written as an XML document [1]. X-definition describes the structure of the processed data elements (we talk about data "models"). X-definition provides validation, processing, or construction of XML documents. An important feature of the X-definition is the ability to process large data [2]. X-definition is an open-source project available at [3].

The source form of an X-definition is an XML document that is compiled into a Java class. From the XML document, it is possible to validate the data and to work with its values, but it is also possible to create XML documents using the X-definition (the so-called construction mode of X-definition). An important feature of the X-definition is the ability to link the process to methods in Java code. The processed XML object can be obtained either as an instance of a Java w3c.dom.Element object or as an instance of a Java X-component object in which the values are accessed by getters and setters (see [4], Chapter 7).

Until version 4.0, X-definition could work with XML data only. Starting with version 4.1 [5], X-definition can also work with JSON data. Since version 4.2, X-definition also allows to describe data as YAML, properties, INI windows, or CSV (comma-separated values).

Extending X-definition to include these data formats requires explicit conversion of JSON, YAML, Properties, Windows INI, and CSV to XML format and vice versa. It is then possible to work internally in X-definition with a format that fits all. This format is called **XON** (**X**-definition **O**bject **N**otation).

2. XML -> JSON conversion

An XML element can be thought of as a JSON map with a single named value whose name is the element name. This value is a JSON array in which the first item is a JSON map containing the attributes of the element, followed by a

sequence of children of the XML element converted to the JSON array items. If the value type of the child node is a text node, it is converted to a JSON string, if it is an XML element, it is converted to a JSON map. If an XML element has no attributes, the first item with the map of attributes may be omitted.

XML format:

```
<product
    productId = "1234"
    productName = "A green door"
    price = "12.50">
  <tags>home</tags>
  <tags>green</tags>
</product>
```

Transformation to JSON format:

```
{ "product" :
  {  "productId" : 1234,
     "productName" : "A green door",
     "price" : 12.50,
     "tags" : [ "home", "green" ]
  }
}
```

3. JSON -> XML conversion

Conversion of JSON data into XML was described by Michael Key at XML Prague [6]. The X-definition uses an algorithm that is more suitable for describing data models of all supported data formats than the Michael Key algorithm.

If the JSON data can be converted to XML (i.e. its structure corresponds to the structure described in the previous paragraph: a map with one named value followed by an array), then the problem is solved. In other cases, XML elements need to be generated that have special elements with the specific namespace "http://www.xdef.org/xon/4.0" (For this namespace the following text uses the nameprefix "jx").

JSON maps are converted into the XML elements "<jx:map>" and JSON arrays into the "<jx:map>" elements. Primitive JSON type values in arrays (strings, numbers, Booleans, and null values} are converted into "<jx:item>" elements, and the values are stored in the form of string to the "val" attribute. Named values of JSON maps are transformed to XML elements with the name which corresponds to the name in a JSON map. If a name does not match the XML NCName syntax (see [7]) it is transformed into a form that matches the syntax of the element name.

The characters of a JSON name which would form invalid XML name are replaced with the sequence "_xh_" (where „h“ is a UTF-16 hexadecimal represen-

tation of the character). The name with an empty string is written as "_x_". So e.g., the name "dogs\tcats" will be converted to "dogs_x9_ cats".

Example of JSON data:

```
{
  "product id" : 123,
  "product name" : "A green door",
  "price" : 12.50,
  "tags" : [ "home", "green" ]
}
```

XML data:

```
<jx:map xmlns:jx = "http://www.xdef.org/xon/4.0">
  <product_x20_id val = "123" />
  <product_x20_name val = "A green door" />
  <price val = "12.50" />
  <tags>
    <jx:array>
      <jx:item val = "home" />
      <jx:item val = "green" />
    </jx:array>
  </tags>
</jx:map>
```

For the source code ergonomics and clarity, if a named value of a map can be transformed to a string it can also be written as an attribute. And also an array of primitive values can be written as a string as in JSON. This is especially useful for writing JSON arrays containing primitive values. Example of writing the simplified form (see the named value "tags" from the example above):

```
<jx:map xmlns:jx = "http://www.xdef.org/xon/4.0"
    product_x20_id = "123"
    product_x20_name = "A green door"
    price = "12.50"
    tags = '[ "home", "green" ]' />
```

4. XON format

The XON format was designed for purpose of writing other types of values than in JSON format. For example, a date in JSON format must be written as a string, the JSON format allows you to write all types of values that can be described in X-definition. XON format can also write value types such as date and time, duration, e.g. email address, and many other values.

Values of the string, boolean or null type are written in XON format the same way as in JSON format. For numeric values, XON enables to write what type of number it is by adding a character after the number (if not specified, the number

is interpreted as long for integer numbers, or double for numbers with a floating-point notation):

Table 1. Number types in XON

Type of number	Character and Java type	Examples
Byte	B (java.lang.Byte)	123b
Short	S (java.lang.Short)	123b
Int	i (java.lang.Integer)	123i
Long	(java.lang.Long)	123
Integer	N (java.math.BigInteger)	123N
Float	f (java.lang.Float)	123f
Double	d (java.lang.Double)	123d or 123.0
Decimal	D (java.math.BigDecimal)	123D

Special values of floating-point numbers:

Table 2. Special values of floating-point numbers in XON

Value of number	type of number	XON format
positive infinity	float (java.lang.Float)	INFf
positive infinity	double (java.lang.Double)	INF
negative infinity	float (java.lang.Float)	-INFf
negative infinity	double (java.lang.Double)	-INF
Not a Number	float (java.lang.Float)	NaNf
Not a Number	double (java.lang.Double)	NaN

Other data types that are supported in X-definition must be described in JSON as strings.However, the notation of other types of values in XON format allows to distinguish the different types by a character at the beginning:

Table 3. Data types in XON

X-definition type	Character	Following notation	XON examples
byte array	b	Base64 notation in brackets	b(N/95BQ==)
char	c	character notation as string length 1	c"x", c"\t", c"\u007"
currency	C	3 capital letters currency in brackets	c(USD)
date, time	d	ISO format od date, time, or DateTime	d2022-06-09T09:00+02:00
duration	P	ISO notation of duration	P1Y1M1DT1H1M1.12S
emailAddr	e	email address as string	e"vaclav <trojan@syntea.cz>"
gps	g	g (GPS parameters in brackets)	g(51.52, -0.09, 55, London)
inetAddr	/	IP ver 4 or IP ver 6 notation	/142.251.1.131 / 1080:0:0:0:8:800:200C:417A
price	p	a decimal number, space, and currency name in brackets	p(12,50 CZK)
telephone	t	notation of telephone number as string	t"+420 123 456 789"
URI	u	notation of URI as string	u"www.xdef.xon/1"

Comments can also be written in XON format to increase readability. Two types of comments are allowed: end-of-line comments starting with the "#" character or embedded comments nested between the "/*" and "*/" characters.

Example of XON notation:

```
# Object with the product Door
{ product :
  {  product id : 123i,
     product name : "A green door",
     price : 12.50D # price in USD
     tags : [ /* list of properties */ "home", "green" ]
  }
}
```

To simplify and shorten the notation, those item names in the maps that match the NCName syntax (i.e. XML name without the colon) can be written without quotation marks (anyway, the value names in maps can always be written enclosed in quotes, just like in JSON format). Thus, the JSON format is a subset of the XON format.

4.1. JSON (or XON) object model in X-definition

The JSON (or XON) model of the object is written as text in the <xd:xon> element, where the model name is in the "xd:name" attribute. Similar to the XML element models in X-definition, the values are described using an X-script, which is always written as a string. Properties of map or array objects can be written using a special element starting with the keyword **x:script,** which must be followed by an equals sign (not a colon!), and the value of the X-script follows as a string value. The following example describes the JSON data model from the previous example:

```
<xd:xon xd:name = "product">
{ products:
  [
    {x:script="occurs *;",
       "productId": "int(100000,999999);",
       "productName" : "string();",
       "price": "decimal();"
       "tags": ["occurs 0..* string();"]
    }
  ]
}
</xd:xon>
```

The result XON data of input data will be:

```
{ "products":
  [
   { "productId": 123456i, "productName": "A green door", "price":
12.50D,
     "tags": [ "home", "green" ]
   },
```

```
    { "productId": 987654i, "productName": "bicycle", "price": 320.00D,
      "tags": [ "ladies bicycle", "silver", "electric" ]
    }
   ]
}
```

(Try to run an example of JSON validation at https:// xdef.syntea.cz/ tutorial/ examples/json.html).

4.2. YAML

Because YAML data is similar to JSON, the YAML data model is described as XON. During processing, the X-definition processor automatically detects whether the data is in JSON, XON, or YAML format. The following data in YAML format will be processed using the same model described above:

```
product:
- product id: 123
  product name: A green door
  price: 12.50
  tags:
   - home
   - green
```

4.3. Properties and Windows INI format

Data in the "properties" format represents a collection of named values. Such data can be easily converted into an XON map with named values. The model of this data can be described as an XON map.

Compared to the properties format, Windows INI data also contains named sections. These can be converted to XON format as named items whose value is a map. See the following example of Windows INI data:

```
TRSUser = John Smith
Authority = CLIENT
MailAddr = jira@synth.cz
[Server]
SeverIP = 123.45.67.8
Signature = 12afe0c1d246895a990ab2dd13ce684f012b339c
```

The model of Windows INI (and also properties) data is written in X-definition as <xd:ini> element:

```
<xd:ini xd:name="TRSconfig">
    # TRS configuration
    TRSUser = string()
    Authority = enum("SERVICE", "CLIENT", "UNREGISTRED");
    MailAddr = emailAddr();
```

```
    [Server]
      SeverIP = ipAddr();
      Signature = SHA1();
</xd:ini>
```

4.4. CSV data

CSV data is a sequence of rows. On each row (line) are written values separated by a comma (or other agreed character). They can be converted to XON format as a two-dimensional array. Missing values in a line are replaced by a null value. In some cases, the first line may contain the names of items in data lines.

The CSV data model can therefore be described as an XON array.

Example of CSV data:

```
Name, Email, Mobile Number
John Brown,john@brown.org,+001 1234565789
Mary, ,+45 987 654 321
Paul, Paul Wiener<paul123@gmail.com>,
```

The model of data above:

```
<xd:xon xd:name = "contacts">
[
   [ "occurs * string();" ], /* header line */
   # one or more lines; where email and telephone are the optional items
   [ $script = "occurs +;", "string();", "? emailAddr();", "?
telephone();" ]
]
</xd:xon>
```

Since the input data may not contain the first row with the description of column names, this circumstance must be communicated at the start of processing (by the boolean parameter skipheader of the cparse methods).

5. Examples of using X-definition

The use of the X-definition takes many forms. The X-definition technology covers the whole spectrum of data integration and implies a consistent approach. Furthermore, The X-definition conceptually allows, for example, creation of automated processes within use cases. Example:

- An analyst creates X-definitions to describe a data structure,
- a programmer dybamically imports the X-definition into a project (for future modifications by the analyst),
- at the beginning of each code compilation, a data model is automatically generated from the X-definition,

- the programmer is already working directly with the data model in a specific language (in our case Java).

What remains is to provide the loading of the input message and its validation. The X-definition library offers a rich interface and can take care of this issue as well. The programmer defacto only needs to choose what type of input the data will be read from (file, stream, string, URL link, etc.).

The following sections describe examples of sub-operations that a programmer or analyst may come across when using X-definition technology.

5.1. Validation mode of processing

Validation mode, as the name implies, is used to validate input data. As such, X-definitions are used to describe the data structure, among other things. The constraints can also be applied to the actual content of the values inside the X-definitions. If we have a particular X-definition (or set of X-definitions) at hand, then we can validate arbitrary data against that X-definition or set of X-definitions.

Example of X-definition and input data that we want to validate using given X-definition.

Validation code X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Example"
root="root" >
  <root a="int();" >
    <b xd:script="occurs *;" >
      ? string();
    </b>
  </root>
</xd:def>
```

XML input data:

```
<root a="123" >
  <b>text</b>
  <b/>
</root>
```

The below Java code sample is somewhat simplified so that the reader is not chalanged by implementation details:

```
String xdef = "Example.xdef";
String xmlData = "Example.xml";

// 1. Compilation of X-definition
XDPool xp = XDFactory.compileXD(xdef);

// 2. Create new instance of XDDocument
XDDocument xdoc = xp.createXDDocument();
```

For the XML data input we also add:

```
Element root = xdoc.xparse(xmlData);
```

The above example deals only with XML data. However, it can also be easily updated by the example below, which generally works for any input data format.

For JSON (or XON) data input:

```
Object o = xdoc.jparse(xmlData);
```

For YAML similarly:

```
Object o = xdoc.yparse(xmlData);
```

Many other X-definition functionalities are built on top of the validation functionality, which internally uses similar code to the above. You can test the data validation functionality against an X-definition at https:// xdef.syntea.cz/ tutorial/ examples/ validate.html. For more details see the Java programming guide with examples at [4].

5.2. External methods

X-definition allows calling external methods implemented in Java. The prerequisite for this is that the method is implemented with static keywords and public visibility. It is also possible to insert input arguments into external methods. The input arguments can also contain actual values from the input data. The call to an external method is called an **X-script**.

The input processing is built on a similar principle as the event-driven architecture. Data processing has a lifecycle. Within this cycle, predefined events (events) are created that have a known order. Basic examples of events:

- *finally* – element processing conclusion,
- *onStartElement* – the start of element content processing (attributes already processed),
- *onTrue* – perform a specific action if the validation of a value was without error.

X-script can be used for multiple purposes. Typically, these are:

- event processing by action,
- advanced value validation,
- creation of a return value within the result (value transform).

In addition, the X-script itself has specific predefined methods that can be used, for example, for additional processing of the input value (example: the *trimText* method - removing white characters at the beginning and end of the string value).

Example of using X-script – without external methods:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="example"
root="root" >
  <root a="int();" >
    <b xd:script="occurs *;
       finally outln('Content: ' + getElementText());">
      ? string();
    </b>
  </root>
</xd:def>
```

Input XML data:

```
<root a="123">
  <b>text 1</b>
  <b>test 2
</root>
```

The output of processing:

```
Content: text 1
Content: text 2
```

Example of implementation of an external method used in an X-definition in Java:

```
com.xdef.example

public class ExternalMethods {
  public static void error (int code) {
  ...
  }
}
```

Example declaration of a Java external method in X-definition:

```
<xd:declaration>
external method void com.xdef.example.ExternalMethods.error(int);
</xd:declaration>

<elem value="required int(0, 10); onFalse error(123);" />
```

External methods allow dynamic validation - for example, calculation of a specific value, previewing a database table, etc.

Example of implementation of an external method used in an X-definition in Java:

```
com.xdef.example

public class ExternalMethods {
  public static boolean tab(String tabName, String colName, String
value) {
    String sql = "SELECT COUNT(*) FROM " + tabName "  WHERE " + colName
```

```
" = " + value;
  ...
  }
}
```

Example of declaration of a Java external method in X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="example"
root="root" >
  <xd:declaration>
    external method
      void com.xdef.example.ExternalMethods.tab(String, String, String);
  </xd:declaration>
  <elem value="required tab('table','column', getText())" />
</xd:def>
```

5.3. Construction mode of processing

Construction mode, on the other hand, serves for creating the XML document according to an X-definition.

While in validation mode the X-definition process the input data according to the models declared in the X-definitions, in construction mode the data is created according to the commands written in the X-script in the "create" section. Thus, an output object can be created and the X-definition serves as a blueprint for its creation. The values in the result object are taken as the result of the command in the "create" section (values from external methods and variables are often used – see next paragraph).

Example:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="example"
root="root" >
  <xd:declaration>
    int num = 0;
  </xd:declaration>
  <root a="date(); create now()" >
    <b xd:script="occurs *; finally outln('getElementText()); create 2;">
      ? string(); create “item: ” + (++count);
    </b>
  </root>
</xd:def>
```

The result will be:

```
<root a="2022-01-01" >
  <b>
    item: 1
  </b>
  <b>
```

```
    item: 2
  </b>
</root>
```

5.4. X-components

X-components are currently a Java-only technology. This technology is similar to JAXB. X-components are used to describe a data model in Java and at the same time, in combination with X-definitions, it is possible to perform bidirectional mapping of input (**data binding**) to/from the X-component model.

One of the main advantages of X-components is the fact that X-components can be **generated** based on the X-definition rule. This implies that based on a blueprint (which can be supplied by an analyst, for example), we can also obtain a data model for the programmer quickly and easily.

The X-components themselves as data models can be further serialized, for example. Thus, X-components can also form, for example, messages that are exchanged between programs. For the description of these messages, we can again use the X-definition rule. Data integration can be solved in this simple way. The only requirement is that all programs involved use the same X-definition rule.

From the above, it shows that X-components are a superstructure over X-definitions and significantly extend the X-definition ecosystem. The combination of these two technologies creates a holistic framework that allows multiple roles (analyst, programmer) to participate in the specification/development of a program while ensuring consistency in the output of the two roles mentioned, which often have different views and requirements on a given issue. These technologies also simplify data integration between programs using X-definition.

The basis for creating an X-component is a special <xd:component> element that extends the X-definition itself. Within this element, we can define which elements we need to generate X-components from. Typically, we generate root nodes this way, and then we can also split internal structures to make the overall resulting model in X-components more granular – often we don't want to work with the whole model, but only parts of it.

To be able to declare what partial data models we need to generate into X-components, we need to know the definition of a position in an X-definition, called XDPosition. Below we give an example of XDPosition values without the need for a deeper understanding of how we arrived at the values. XDPosition can be imagined as an alternative to Xpath. For this reason, the example is trivial, so that the reader has a chance to orient himself quickly. The XDPosition values are written on the right side only for the demostration purpose.

Example of XDPosition values in X-definition:

```
<xd:def name = "Model">
  <A>                       Model#A
```

```
    <B                    Model#A/B
      b = "string();"/>    Model#A/B/@b
  </A>
</xd:def>
```

With the above knowledge, we are already able to declare the required X-components. Example of X-definition enriched with X-component declaration:

```
<xd:def
  xmlns:xd="http://www.xdef.org/xdef/4.1"
  xd:name="Vehicle"
  xd:root="Vehicle">

  <Vehicle VIN = "required string();" />

  <xd:component>
    %class cz.syntea.tutorial.Vehicle
      %link Vehicle#Vehicle;
  </xd:component>

</xd:def>
```

The above example is trivial - for this reason, we are only interested in the root element. We use the %class keyword to specify in which qualified path the X-component should be created. The next keyword %link specifies the position within the X-definition (the XDPosition value).

Example of a generated X component (from the previous example):

```
package cz.syntea.tutorial;

public class Vehicle implements org.xdef.component.XComponent {

  public String getVIN() { return _VIN; }
  public void setVIN(String x) {_VIN = x;}

  private String _VIN;

  // Constructors and implementation of the interface XComponent ...
}
```

X-components support a range of keywords, such as support for interface creation, references across the X-definition set (in practice we work with sets), enumeration, aliases for models and attributes, etc.

If we have an X-component created in this way, we can perform input validation and data binding in a very similar way to the validation itself. Below is a simplified example of populating the X-component model with XML input data so that the reader is not chalanged by implementation details.

```
String xdef = "vehicle.xdef";
String xmlData = "vehicle.xml";

XDPool xp = XDFactory.compileXD(xdef);
XDDocument xdoc = xp.createXDDocument();
XComponent xc = doc.parseXComponent(source, Vehicle.class);
Vehicle vehicle = (Vehicle) xc;
```

The example is very similar to the validation mode, the only difference is the last two lines, where instead of validation itself, we require data binding.

5.5. Localization of data

X-definition have the functionality of localizing the data model prescription itself, called X-lexicon. X-lexicon is suitable, for example, for situations where different customers require localization of data-identical messages into their language. The data model itself only needs to be defined once, and the reflecting data model translations need to be created also once. To create the translations, we again use the knowledge of XDPosition.

The X-lexicon definition itself consists of a special <xd:lexicon> element that extends the X-definition itself. Within this element, we define the required individual localizations.

Example of a common prescription in X-definition (names in English):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.1" xd:name = "contract">
  <Contract
    Number = "required int();">
    <Owner
      Name      = "required string();"/>
  </Contract>
  ...
</xd:def>
```

Example of localization for German language:

```
<xd:lexicon xmlns:xd="http://www.xdef.org/xdef/4.1"
language="Contract_deu">
  contract#Contract                Vertrag
  contract#Contract/@Number        Nummer
  contract#Contract/@Date          Datum
  contract#Contract/Owner          Inhaber
</xd:lexicon>
```

6. Example of processing of JSON data

In this chapter, we will show how to process JSON data using X-definition. The input data includes data from telecommunication network points, described in the following X-definition:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.1' name="net"
root='netdata'>

<xd:declaration>
   external method {
     boolean exc.base.Network.checkId(XXData xnode);
     void exc.base.Network.setId(XXData xnode);
     /* There are in the a real project more external methods... */
   }

   type id string(%length=35);
   /* There are in the a real project more validation types ... */
</xd:declaration>

<xd:xon name='netdata'>
{
  "container":            "boolean();",
  "res-id":               "checkId(); onTrue setId(getParsedResult());",
  "roles": [
      "occurs * enum('OLT','MXU','OTHER');"
  ],
  "mac":                  "optional id(); onTrue setId();",
  "ref-parent-subnet":    "optional id();",
  "dev-sys-name":         "optional string();",
  "communication-state":  "optional int(0,1);"
  "remark":               "optional string();",
  "is-gateway":           "optional int(-1,1);",
  "ip-address":           "optional ipAddr();",
  "admin-status":         "optional enum('active', 'inactive');",
  "location":             "optional string();",
  "physical-id":          "optional int(1, *);",
}
</xd:xon>

</xd:def>
```

Input JSON data:

```
{ "container":false,
  "res-id":"fda88901-3b7f-33ec-80f260a95523fd7c",
  "roles":["MXU"],
  "communication-state":1,
```

```
  "remark":"",
  "is-gateway":-1,
  "ip-address":"172.25.39.18",
  "admin-status":"inactive",
  "location":"",
  "physical-id":1
}
```

The Java program (simplified) for process JSON data as described above:

```
package exc.base;

import java.io.File;
import java.net.InetAddress;
import java.util.Map;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDParseResult;
import org.xdef.XDPool;
import org.xdef.sys.ArrayReporter;
import org.xdef.xon.XonUtils;

public class Network {

    public static XDParseResult checkId(XDParseResult id) {
        if (id.matches()) {
            // String s = id.getParsedValue().toString();
            // you can check database integrity etc.
            // and add error report an error with id.error("error text");
        } else {
            // It was found an error in the validation method
            // You can handle this situation. However, a stardard error
            // will be reported.
            System.err.println("ID is not correct: " +
id.getParsedString());
        }
        return id;
    }

    public static void setId(XDParseResult id) {
        String s = id.getParsedString();
        // set id e.g. to a database.
    }

    public static void main(String[] args) {
        File xdef = new File("data/network.xdef"); // X-definition
        File input = new File("data/network.json"); // JSON data
```

```
            XDPool xpool = XDFactory.compileXD(null, xdef);
            XDDocument xdoc = xpool.createXDDocument("net");
            ArrayReporter reporter = new ArrayReporter();
            Object o = xdoc.jparse(input, reporter);
            if (reporter.errors()) {
                System.err.println(reporter.printToString());
            } else {
                // parsed Xon object is instance of java.util.Map
                Map xon = (Map) o;
                // // the item ip-address in XON is java.net.InetAddress
    object
                InetAddress ipAddr = (InetAddress) xon.get("ip-address");
                System.out.println("OK, ipAddr = " + ipAddr);
                System.out.println("Parsed XON data:");
                System.out.println(XonUtils.toXonString(xon, true));
            }
        }

    }
```

7. Conclusions

- Support for XML, XON, JSON, YAML, Properties, Windows INI, and CSV formats allows you to use X-definition in projects that work with all these source formats.
- X-definition describes in detail the data structure in all supported formats (XML, JSON, YAML, Properties, Windows INI, CSV, and XON). They can thus replace XML Schema, JSON Schema, etc.
- X-definition allows processing of unlimited size of data of all supported formats.
- X-definition supports error handling and a detailed error log.
- As a result of processing an XON object can be obtained. This object contains parsed values converted to Java data types.
- As a result of processing, an X-component object can be obtained. X-component is an instance of a Java object, where processed data are accessible with getter and setter methods.
- Xon data format is suitable for exchanging data between remote systems.

Bibliography

[1] Trojan, Václav: *XDefinition* . XML Prague, Prague, 2005

[2] Selak, C: *Extracting Data From Very Large XML Files With X-definition* . 31. May 2022. https://dzone.com/articles/extracting-data-from-very-large-xml-files-with-x-d

[3] *X-definition* https://github.com/syntea/xdef

[4] Trojan, Václav: *X-definition 4.1, programming guide* . 2021. https://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.1_Programming.pdf

[5] Trojan, Václav: *X-definition 4.1, language description* . 12. July 2021. https://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.1.pdf

[6] Key, Michael: *Transforming JSON using XSLT 3.0* . 2016. https://www.saxonica.com/papers/xmlprague-2016mhk.pdf

[7] Bray, Tim - Hollander, Dave - Layman, Andrew - Tobin, Richard: *Namespaces in XML 1.1* . W3C Recommendation 16 August 2006. https://www.w3.org/TR/xml-names11/

A Pilot Implementation of ixml

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[7], [8], [9], [10], [11], enabling authors to write documents and data in a format they prefer while providing XML for processes that are more effective with XML content. By the time of the publication of this paper, it is anticipated that the official version of ixml [12] will have been announced by the ixml working group.

During the development of ixml, a pilot implementation was built in order to support decisions on the development of the notation, and to provide examples of the output ixml produces.

This paper describes the implementation, decisions taken, and how certain processes work, such as serialisation, and dealing with ambiguity, and ends by discussing future work to be done.

Keywords: ixml, markup, XML, implementation, notation design, grammars, parsing, earley

1. Invisible XML

Numbers are abstractions: you can't point to the number three, just three apples, or three sheep. Three is what those apples and sheep have in common.

You can represent a number in different ways, 3, III, 0011, 三, ३, ੩, ૩, ৩, 〣, ୩, ౩, ೩, ൩, III, ௩, ༣, ፫, ვ, trois, drie. You can concretise numbers as a length, a weight, a speed, a temperature.

But in the end, they are all the same *three*.

The idea behind ixml is that data too is an abstraction, which we are often obliged for different reasons to represent in some way or another. But in the end those representations are all of the same abstraction. It is worth noting that http recognises this too, with content negotiation around a URL: the URL represents a single resource; content negotiation allows the selection of a particular representation of that resource [6].

Ixml takes representations of data, typically with implicit structure, recognises that structure, and forms a representation where the structure is made explicit.

Some representations are weaker than others: they may not be able to faithfully represent all of the abstraction, and are therefore not reversible, but XML is

probably the best available general notation for approaching the representation of any abstraction.

The intention behind ixml is to allow extracting abstractions from representations; of converting weaker representations of abstractions into more explicit representations, with XML therefore an excellent target for that.

2. Processing

An ixml processor takes a a document in a particular (textual) format, along with a description of that format, in the form of a grammar, and uses it to parse the document. This produces a structured parse tree of the document, which can then be processed in a number of ways, the primary one being serialization as XML.

Figure 1 illustrates this: the circle is the processor, a square represents a textual representation of a document, and a triangle a structured representation. The documents to the left and right of the processor are the same document, but in different representations.

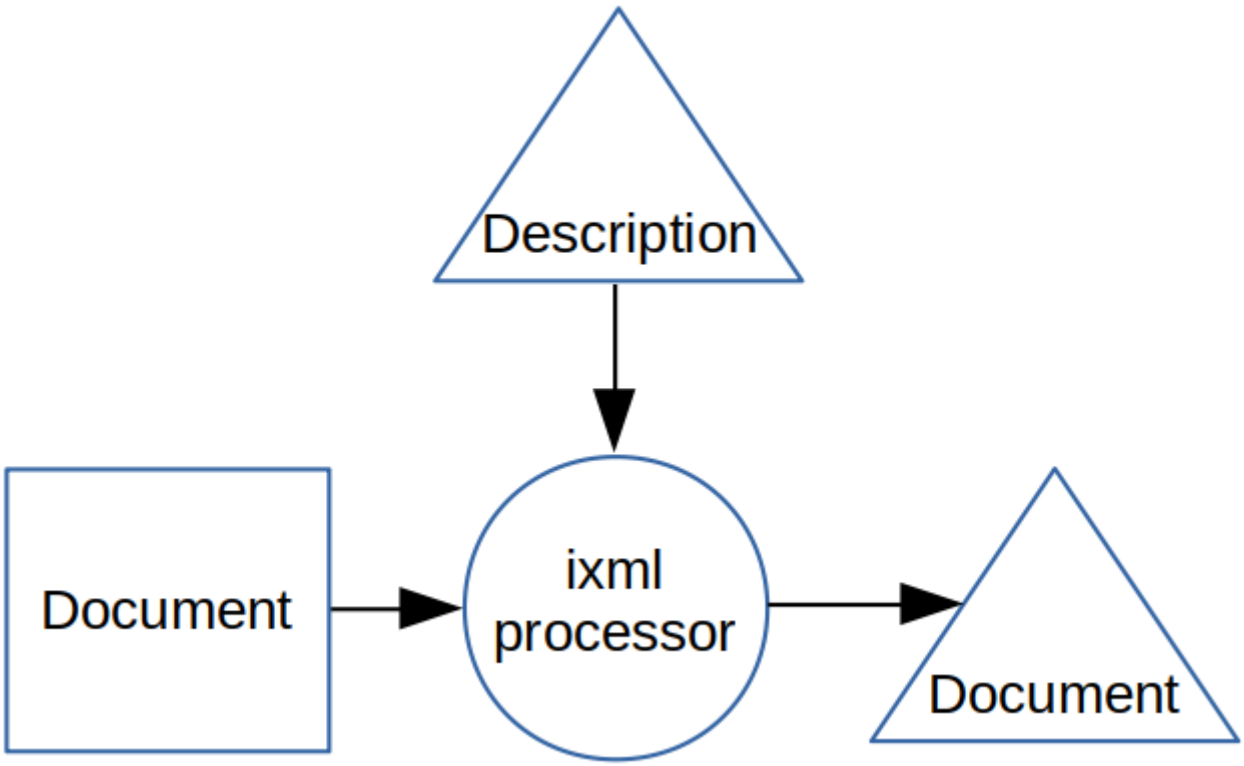


Figure 1. Fig. 1: ixml processing step

As you can see, the format description is drawn as a structured document. However, since it is normally supplied as an ixml document in textual form, this also has first to be processed, in exactly the same way, by the ixml processor, but using a description of the ixml format. This results in the structured version of the

description. Figure 2 illustrates this. As you can see there is a also a presumed bootstrap stage that produces the initial structured version of the description of ixml itself, where the structure of ixml is initially presumed during the bootstrap.

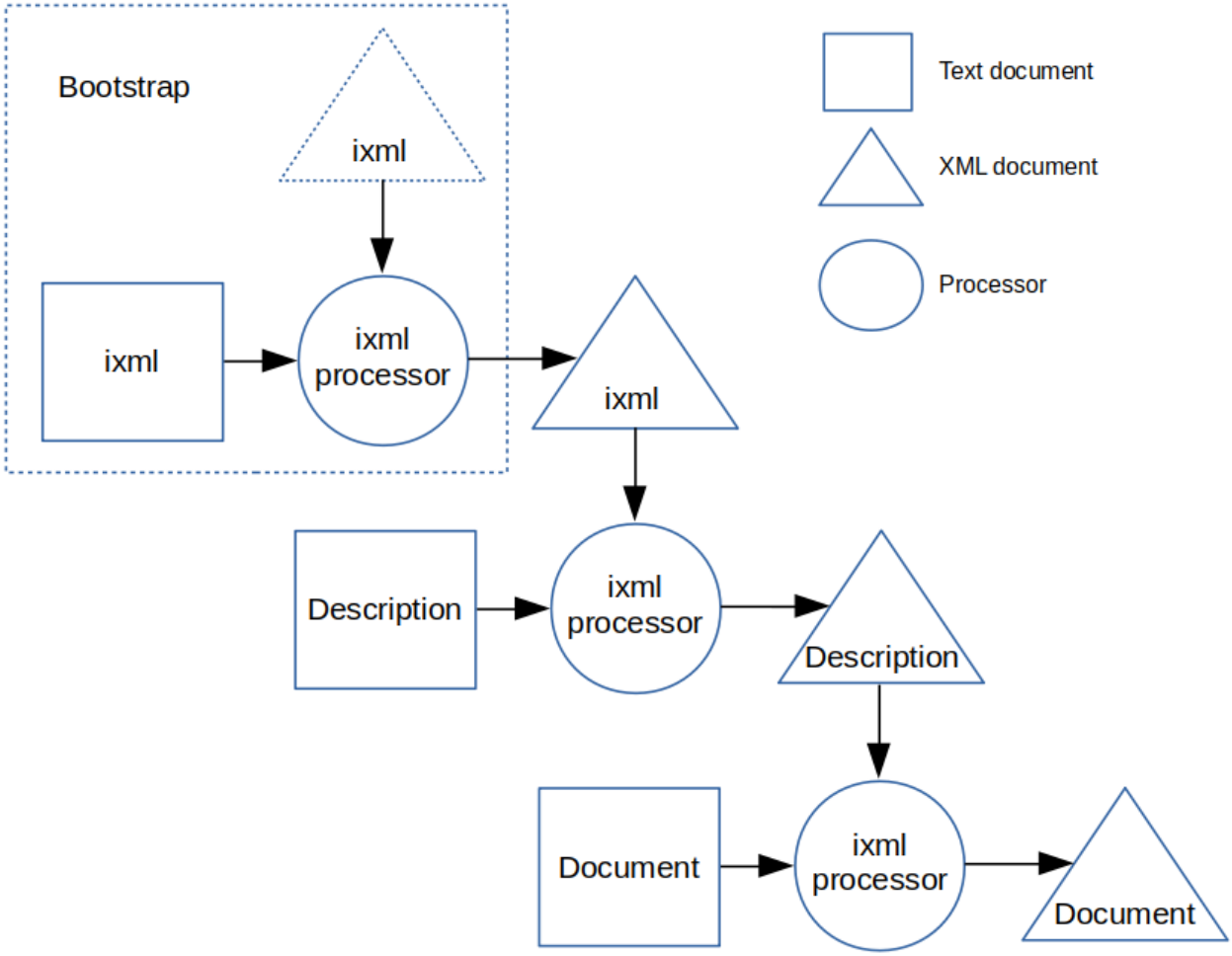


Figure 2. Fig. 2: The complete processing cycle

3. How ixml works

An ixml description is a grammar consisting of a series of rules. A rule consists of a name, and a number of alternatives separated by semicolons:

```
statement: assign; call; if; while.
```

Alternatives consist of a sequence of zero or more terminals and nonterminals separated by commas:

```
assign: id, ":=", expr.
expr: id; number.
```

Input that matches the grammar is parsed into a parse tree, which is then serialised as XML. So for input "`i:=0`", you would get

```
<statement><assign><id>i</id>:=<expr><number>0</number></expr></assign></
statement>
```

So-called *marks* can be added to rules to affect the serialization. Rules can be marked to be serialised as attributes:

```
assign: @id, ":=", expr.
```

which would give

```
<assign id="i">:=<expr><number>0</number></expr></assign>
```

terminals can be marked to be deleted:

```
assign: @id, -":=", expr.
```

which would give

```
<assign id="i"><expr><number>0</number></expr></assign>
```

as can nonterminals:

```
assign: @id, -":=", -expr.
```

which would eliminate the enclosing `<expr>` tags around its element content:

```
<assign id="i"><number>0</number></assign>
```

A recent addition to ixml are *insertions* which allow characters to appear in the serialisation that weren't in the input:

```
assign: @id, -":=", -expr, +";".
```

which would give

```
<assign id="i"><number>0</number>;</assign>
```

4. Implementation

4.1. Parsing algorithms

There are many known parsing algorithms, and most have restrictions of one sort or another on the classes of language that they recognise, and include restrictions on the languages, or the grammars, or both, for them to work.

For example, LL(1) grammars require that when a grammar rule has alternatives (as most do), like

```
sentence: a; b; c.
```

that the choice of whether to parse the sentence as an `a`, `b`, or `c` must be decidable by looking at most one token ahead. To achieve this, languages typically must go through an initial stage prior to parsing to split them into individual tokens, which in turn means that a grammar has to be accompanied by a description of how tokens are formed.

All this makes life difficult for the grammar writer, who must not only know the rules for the parsing algorithm, and how to apply them, but must write two descriptions, one for the grammar, and another for the tokens.

To avoid these problems, ixml requires implementations to use a general parsing algorithm, without extra restrictions, and without the need to specify tokens. Examples of such algorithms are [3], [14], [2], [5], and [4].

4.2. Parsing

The pilot implementation of ixml uses Earley as one of the earliest and best-known of the general parsing algorithms.

As pointed out in [9], Earley can be seen as a pseudo-parallel parsing algorithm, where when it has to parse a rule like

```
sentence: a; b; c.
```

it splits into three parallel sub-parsers to parse the three alternatives starting at the same point in the input. If a sub-parse fails at any point, it terminates without further ado; if it succeeds, it records its sub-parsetree(s), and terminates.

4.3. Serialisation

Once parsing is finished, what remains is a so called parse-forest, a collection of linked sub-parse-trees.

The first action is to see if the parse has been successful, by looking if there is a successful parse node for the root symbol that starts at the first character position and ends at the last. If so then serialisation can begin.

Serialisation is a question of doing a tree walk: non-deleted nonterminals are serialised to XML elements, deleted nonterminals just have their children serialised, non-deleted terminals (and insertions) are just output. There is an additional elaboration for nonterminals serialised as attributes, since they come before element content, and so for any non-deleted nonterminal, you have to do one walk for the attributes, and then one for the content.

Because of element deletions such as

```
assign: -target, -":=", expr.
target: @id.
```

since the element that the id attribute is ostensibly on is deleted, the attribute has to move up to the nearest non-deleted element:

```
<assign id="i"><expr>...</expr></assign>
```

so the tree walk for attributes has to look not only at the level of the (undeleted) element, but also recursively within deleted sub-elements.

4.4. Ambiguity

The parse may have been ambiguous: that is, it satisfied the rules of the grammar in more than one way. The serialisation tree-walk is not going to care about this, and as long as the parse has been successful it will produce a serialisation of one of the parses.

However, it is usually important that the consumer of the serialisation know that the serialisation is only one of the possible cases. To this end, ixml requires an `ixml:state="ambiguous"` attribute to be added to the root element of the serialisation to signal that fact. This involves a simple initial tree-walk to discover if any route from the top node is ambiguous.

5. The Pilot Implementation in Use

The pilot implementation was originally written to support the development of the language, to try out different approaches and solutions, as well as to generate example outputs to support papers on the language.

The primary aim at that stage was therefore speed of implementation and flexibility, and not to create an industrial-strength, top-speed implementation, since it was too early at that stage while the language was constantly being altered. Consequently it was written in an interpreted language with very-high-level data types, ABC [1], in about 500 lines of code for the bootstrap parser, and 700 for the full processor.

It was later also used to support an ixml tutorial [13]. After an earlier experience with a tutorial for a different language, where the tutees had to install software themselves to run the examples, it was decided for ixml instead to supply an online processor, where the example ixml grammar and its input were submitted to the processor, and the resulting XML would be returned. A problem was that the pilot implementation was non-reentrant, and so the solution was to store the grammars and input in files, and serve them one by one to the processor, while the server busy-waited for the result file.

6. Future Work

6.1. Serialising to memory

Although the processing diagram above suggests that ixml always serialises its input documents to XML, there are other options. In particular since the format-description document is used as input in the next step to parse the final document, it can be more efficient to serialise the format description straight to memory, into the form required by the parser for representing grammars. This also allows for some simplifications, in particular because the grammar for ixml

will never produce ambiguous parses, and so doesn't need the two-passes otherwise necessary for serialisation. This of course also speeds up processing, since it eliminates one whole parsing phase.

6.2. Round-tripping

An ixml grammar can be seen as a function mapping one representation on to another. In simple cases, such as a grammar with no deletions and no attributes, mapping the output back to the input form is trivial, since it just involves concatenating the element contents. This is because the default is for all input characters to be copied to the output serialisation, and in simple cases like this all that happens is content gets enclosed by element tags to reveal the underlying structure. Remove the tags, and you have the input again.

Deletions complicate matters. If there are only element deletions, then it remains trivial, because all input characters are still in the output; only some element tags have been omitted, so concatenating element content still works.

However, with terminal deletions, characters are lost that have to be restored. To deal with this, we have to parse the serialisation using the grammar that produced it, with a similar parser to Earley, in order to discover which characters have been deleted.

To parse a (non-deleted) nonterminal, you must expect the start tag for that rule. For instance, for

```
statement: assign; call; if; while.
```

you expect

```
<statement>
```

and then initiate four parallel sub-parsers, one for each of the alternatives, which to succeed must also be followed by the terminating tag `</statement>`.

To parse a deleted nonterminal, you just initiate the sub-parsers.

To parse a (non-deleted) terminal, you must just expect the same string in the element content at the current point.

To parse a deleted terminal, you match zero characters, and insert the string in the parse. Interestingly enough, this is exactly how an insertion works when parsing in the other direction. And indeed to parse an insertion, you have to expect the characters in the string in the same way as a non-deleted terminal, but insert nothing in the parse, just like a deletion in the other direction.

The other challenge is dealing with attributes. Because of how attributes are placed on XML elements, and additionally due to deleted nonterminals as explained above, attribute content can appear earlier in the serialisation than the content that preceded them in the input. Therefore attributes have to be held in abeyance, as separate input streams, until the point in the parse where they appear in the grammar. Then the input must come from the serialisation of the

attribute, and all sub-rules must be treated as if deleted, until the end of the rule forming the attribute, which by then must have consumed all of the input coming from the serialisation of the attribute.

The result of this process will be a (potentially ambiguous) parse tree.

For a serialisation like

```
<assign id="i"><number>0</number>;</assign>
```

being parsed against

```
assign: @id, -":=", -expr, +";".
```

you will get a parse tree like

```
<assign><id>i</id>:=<expr><number>0</number></expr></assign>
```

from which you can concatenate the element content to give "`i:=0`".

However, for a grammar that includes a rule like:

```
term: id; number; -"{", expr, -"}"; -"(", expr, -")".
```

where expressions may be bracketed with either {} style or () style brackets, both of these latter two alternatives, because of the deletions, will produce exactly the same serialisation, namely:

```
<term><expr>...</expr></term>
```

and so parsing the serialisation back we will get two successful parses:

```
<term>{<expr>...</expr>}</term>
```

and

```
<term>(<expr>...</expr>)</term>
```

in other words, an ambiguous parse. So, just as with serialisation, we have to choose one. In other words, because the serialisation throws away the information about which brackets were used, we can't guarantee to round-trip the serialisation perfectly.

Similarly, if a serialisation deletes all spaces or comments in the input, there is no way to recreate them. They are lost in the round-tripping.

6.3. Translating to other languages

The main shortcoming of the pilot implementation is that it is written in a programming language that doesn't fully support Unicode. While it is possible to parse Unicode documents with it, any feature of ixml that requires accessing *properties* of Unicode characters, such as hexadecimal representations, ranges, or character classes, can't be fully supported.

To this end, future work will include translating the implementation to one or more other languages.

7. Conclusion

Designing a notation requires many aspects to be taken into account simultaneously, such as usability, functionality, and ambiguity. Having an implementation that is easily modifiable during design of the notation is almost essential for good progress.

The current pilot implementation has served well, and while other implementations are now emerging, it will probably be retained for future design research.

References

[1] Leo Geurts et al.. *The ABC Programmer's Handbook*. Prentice-Hall 1990. 0-13-000027-2. https://www.cwi.nl/~steven/abc/programmers/handbook.html .

[2] Itiroo Sakai. *Syntax in universal translation*. In 1961 International Conference on Machine Translation of Languages and Applied Language Analysis 1961. 593–608. https://aclanthology.org/www.mt-archive.info/50/NPL-1961-Sakai.pdf .

[3] J. Earley. *An efficient context-free parsing algorithm*. In Communications of the ACM 13(2) February 1970. 94–102. 10.1145/362007.362035.

[4] ElizabethAdrian ScottJohnstone. *GLL Parsing*. In Electronic Notes in Theoretical Computer Science Volume 253.7 17 September 2010. 177-189. 10.1016/j.entcs.2010.08.041.

[5] Masaru Tomita. *Generalized LR Parsing*. Springer Science & Business Media 1991. 978-1-4615-4034-2. 10.1007/978-1-4615-4034-2.

[6] R. Fielding et al.. *Hypertext Transfer Protocol -- HTTP/1.1*. IETF 1999. https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html .

[7] Steven Pemberton. *Invisible XML*. Proceedings of Balisage: The Markup Conference 2013 Balisage Series on Markup Technologies vol. 10 2013. 10.4242/ BalisageVol10.Pemberton01.

[8] 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 .

[9] Steven Pemberton. *Parse Earley Parse Often: How to Parse Anything to XML*. In Proc. XML London 2016 2016. 120-126. http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120 .

[10] Steven Pemberton. *On the Descriptions of Data: The Usability of Notations*. In Proc. XML Prague 2018 2018. 143-159. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf .

[11] Steven Pemberton. *On the Specification of Invisible XML*. Proc.XML Prague 2019 pp 413-430. https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf#page=425 .

[12] Steven Pemberton. *Invisible XML Specification*. invisiblexml.org 2022. https://invisiblexml.org/ixml-specification.html .

[13] Steven Pemberton. *Hands On ixml*. Declarative Amsterdam 2021. https://declarative.amsterdam/show?page=da-tutorial-ixml-2021 .

[14] S.H. Unger. *A global parser for context-free phrase structure grammars*. In Communications of the ACM 11(4) April 1968. 240–247. 10.1145/362991.363001.

8. Postscript

In this author's experience, the hardest part of getting an article into Docbook format (the format used by this conference) is getting the bibliography right. To this end, the above bibliography was produced with the help of ixml. For instance, the text

```
[spec] Steven Pemberton (ed.), Invisible XML Specification,
invisiblexml.org, 2022, https://invisiblexml.org/ixml-
specification.html
```

was processed by an ixml grammar whose top-level rules were

```
bibliography: biblioentry+.
biblioentry: abbrev, (author; editor), -", ", title, -", ", publisher,
-", ", pubdate, -", ", (artpagenums, -", ")?, (bibliomisc;
biblioid)**-", ", -#a.
```

yielding

```
<biblioentry>
 <abbrev>spec</abbrev>
 <editor>
 <personname>
       <firstname>Steven</firstname>
       <surname>Pemberton</surname>
 </personname>
 </editor>
 <title>Invisible XML Specification</title>
 <publisher>invisiblexml.org</publisher>
 <pubdate>2022</pubdate>
 <bibliomisc>
 <link xlink:href='https://invisiblexml.org/ixml-specification.html'/>
 </bibliomisc>
</biblioentry>
```

which can further be tweaked by hand.

Expression Elaboration

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper describes an approach to evaluation of expression-based languages such as XSLT, XQuery, and XPath, in which nodes on the expression tree output by the language parser are converted to lambda expressions in Java, Javascript, or C#, with the aim of doing as much work as possible once only, in advance of the actual expression evaluation.

1. Introduction

Traditionally, when processing a language such as XSLT, XQuery, or XPath, there is a choice of two approaches: interpretation, or code generation.

In its pure form, interpretation works by constructing a parse tree of the source code, and then writing an interpreter that evaluates the constructs on this parse tree, typically in bottom-up fashion: a node on the tree is evaluated by first evaluating its operands (represented as child nodes on the expression tree), and then combining the results according to the semantics of the relevant operator.

In practice it is possible to improve the performance of the interpreter by analyzing and modifying the expression tree before evaluation starts: examples of these processes include resolving references (such as references to variables and functions), inferring types, and optimizations such as loop-lifting (pulling code out of a loop to avoid repeated execution). Declarative languages like XSLT, XQuery and XPath benefit greatly from such optimisations.

By contrast, code generation in its pure form takes the parse tree and converts it into a sequence of machine instructions that are then executed to evaluate the program. Today, to achieve portability, these will normally be instructions for a virtual machine (such as the Java VM) rather than physical hardware.

In practice the two approaches are not quite as distinct as it might appear, and it's certainly possible to use a blend of both. In particular, even when code generation is used, much of the generated code will consist of calls into a run-time library.

Both approaches have been used in the Saxon product. When code-generation was first introduced, it often delivered a performance boost of the order of 25% (though the range was anything from 0% to 50%). However, there was a penalty: compile time costs increased. Given that in many workloads, stylesheets are compiled every time they are executed, this turned out to be a poor trade-off; it is

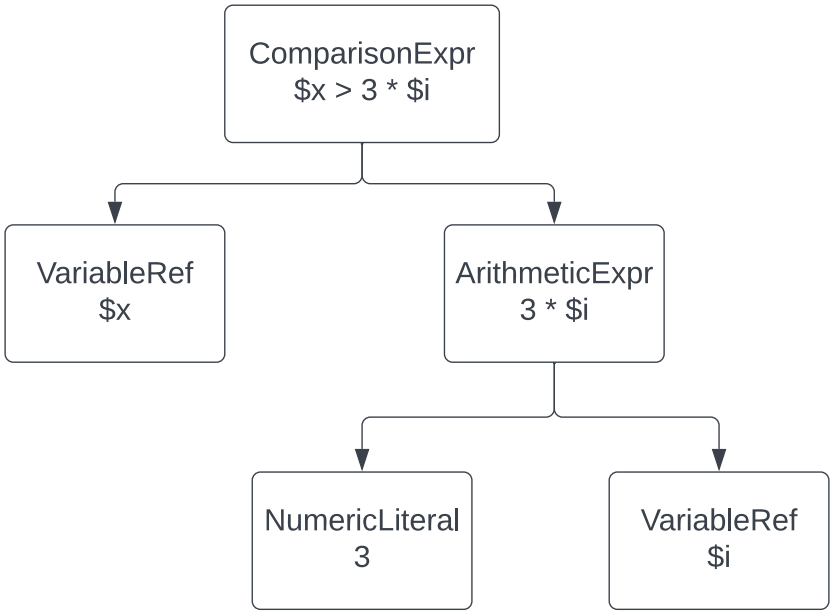
quite common for compilation costs to exceed run-time costs by an order of magnitude.

It's also noticeable that the benefits of generating Java byte-code have declined over time. It's hard to be certain about the causes of this, but we suspect it's primarily because the Java hot-spot compiler has improved over time to the extent that it can often speed up our interpreted code to make it just as fast (or sometimes faster) than the code that we laboriously generate ourselves. That may be partly because the bytecode that Saxon generates is rather different from the bytecode that the Java compiler generates, and the hot-spot JVM (naturally) is optimised for the latter. It may also have something to do with locality of reference: in a modern CPU, the main bottleneck is not the speed of executing instructions, but the speed of moving data from main memory into the CPU cache, which in turn benefits substantially if all the data (and code) needed to execute some function is in nearby storage locations, so they can be transferred to the CPU en bloc, thus improving CPU cache hit rates. Generating code that takes advantage of these low-level effects is a specialized skill, and it's not surprising if the team working on the hot-spot compiler are better at it than we are.

In the Javascript version of Saxon (SaxonJS) we developed an alternative approach to expression evaluation, which has proved very successful. We call this "elaboration" (a term borrowed from Algol68, though we don't claim to use it with precisely the same meaning.) This approach relies heavily on the fact that the languages we are dealing with are side-effect free, which gives us a lot more freedom in rearranging how code is executed. We've recently been extending its use to the Java and C# versions of the Saxon product, and this paper reports preliminary results of this work. This paper describes how expression elaboration works. But first we'll look in more detail at how the two existing strategies, the interpreter and the bytecode generator, are implemented, and at their strengths and weaknesses.

2. The Expression Interpreter

At static analysis time, Saxon parses the source XSLT or XQuery code and generates an expression tree: a hierarchic structure of Java objects in which each node represents an expression (or another construct, such as an XSLT instruction, or a clause in a FLWOR expression), with links to its subexpressions. For example, an expression like `$x > 3 * $i` produces a `ComparisonExpr` node, with two child nodes, a `VariableReference` node for `$x`, and an `ArithmeticExp` node for subexpression `3 * $i`; this in turn has two child nodes for its operands (a `NumericLiteral` and a `VariableReference`. Each kind of node is represented by a subclass of the `Expression` class, and has additional fields for relevant properties such as the arithmetic operator, the name of the variable, and the value of the literal.



In principle, every kind of `Expression` overrides the method `Expression.evaluate(Context context)` which takes as its argument the current evaluation context (providing access to details such as the context item, position, and size, the current group, current mode, and so on). Calling this method returns the result of evaluating the expression, which in general is a `Sequence` object.

(The above is a simplification. We actually provide a variety of evaluation methods: `iterate()` for lazy evaluation as an iterator, `process()` for push mode evaluation where the result are written to a serializer rather than being returned to the caller, `evaluateItem()` where the result is known to be a singleton, and so on).

The expression tree originates from parsing of the source XSLT, XPath, or XQuery code, but before we get to evaluate it, it goes through a number of modifications. For example:

- Nodes are added to the tree to represent implicit operations such as type checking, type conversion, and sorting of nodes into document order. To achieve this, type analysis first annotates the tree with information about the expected type of each construct.
- Links are added, for example from a variable reference to the corresponding variable declaration, or from a function call to the function declaration.
- Operational information is added to the tree, for example local variables are allocated a slot number in the stack frame for their containing function or template, so that reading and writing of variables at run-time can use simple numeric addressing, rather than matching of user-oriented variable names as strings.

- Expressions are optimized using local tree rewrites. Some expression kinds are used which can only result from such optimization rewrites: an example might be an `IntegerRangeTest` with three operands representing the expression `V = P to Q`, which tests whether the value of `V` is in the range `P` to `Q` inclusive. Other rewrites generate constructs that could have been written by the user explicitly, for example `P = Q` might be rewritten as `P eq Q` if it is known (from type analysis) that both operands are singletons. The more powerful optimizations change the structure of the tree, for example by moving a subexpression out of a loop where it is safe to do so.

The design of the expression tree has some limitations:

- The same data structure is used during static analysis and at run time. In principle much of the information that's needed for static analysis could be discarded once evaluation starts; evaluation might benefit from a lighter-weight structure designed explicitly for that purpose.
- The data structure is, for most purposes, read-only at run-time. That's necessary for thread safety - if you process several source documents concurrently in a web server using the same stylesheet, they will share the same copy of the expression tree. This means it's not possible to do things like replacing a global variable reference with the value of the variable once the value is known. (Actually, it's not completely read-only. There are some changes that happen when a node in the tree is evaluated for the first time, for example. Such operations need careful attention to thread safety.)

3. Bytecode Generation

The Enterprise Edition of Saxon attempts to speed up execution by generating JVM bytecode for evaluation of selected parts of the expression tree. Because bytecode generation is itself an expensive process, and may consume large amounts of memory, this is done very selectively. During static analysis, Saxon identifies particular expressions as candidates for bytecode generation. The body of a function or template is always such a candidate, but so are smaller units of code such as the predicate in a filter expression, or the body of an `xsl:for-each` instruction.

This isn't as sophisticated as the JVM hot-spot compiler, which actually monitors how effective its optimizations are, and is capable of reversing them if they prove not to be worthwhile. But it's the same general idea.

There are two interesting questions to ask about bytecode generation: how effective is it, and why?

The answer to the first question is that we see bytecode generation speeding up XSLT and XQuery execution by anything from 0 to 25%; but most cases, sadly, are towards the lower end of that range. It's most effective with simple queries

dominated by evaluation of a simple predicate — but it has to be one that can't be optimized by other techniques such as indexing. For example, in the XMark benchmark suite, query Q11 execution improves from 145ms to 115ms with bytecode generation enabled (around 20%). This query is dominated by the execution of a single predicate of the form `[$vv > 5000 * data(.)]`. Here `$vv` is a local variable generated by the optimizer (as a result of loop-lifting an expression out of the predicate), and `data(.)` involves atomizing a node and converting its string value to an `xs:double`. In fact, string-to-double conversion dominates the query execution time. This is done in a library routine, and there's no opportunity for bytecode generation to speed it up.

By contrast, we found that execution of a large DocBook XSLT transformation improves only from 10.17s to 10.08s as a result of bytecode generation. We've profiled this, and it's very hard to identity significant hot-spots that account for a substantial part of the total execution time.

Where exactly does bytecode generation help? It's surprisingly difficult to answer this question.

It's easy to see where it doesn't help: most of the run-time execution is spent doing things like string-to-number conversion, regular expression processing, navigation of the TinyTree data structure, parsing, and serialization, where the logic is all in library routines that are exactly the same whether invoked by the interpreter or by generated bytecode. So where do we get gains? I think the answer is some combination of the following:

- Reduced navigation of the expression tree. Some of the expressions on the expression tree execute so quickly that finding your way to the expression that needs to be evaluated is as much work as doing the actual evaluation. This is pure overhead in the interpreter.

- Eliminating run-time checks. Even with interpreted code, we go to great lengths to do everything we can at compile time to reduce work done at run-time. For example, if a regular expression or a collation URI is supplied as a string literal, we'll always try to take advantage of the fact. And we do static type analysis to avoid unnecessary run-time type checking. But sometimes it's just not practical. For example, there are many instructions where there's a run-time error if the context item is absent, or if it isn't a node. We might know at compile time that this check isn't needed on a particular path, but with the interpreter, it's simplest to do it anyway. The bytecode generator can be a bit more selective and avoid a few unnecessary instructions.

- Inlining. When we generate code for a predicate like `[$vv > 5000 * data(.)]`, the code all goes in a single generated method. There are no calls from the method that does the comparison to the method that does the arithmetic to the method that does the atomization. Fewer method calls means less

overhead; it also means that the next instruction you want to execute is more likely to already be in the CPU cache.

Now, you might ask, surely the JVM hot-spot compiler can do inlining anyway, so why do we need to do it ourselves? Well, there's a very important difference. In the Saxon interpreter, the methods are highly polymorphic ("megamorphic" is the term used by the JVM experts). That is, we have literally a couple of hundred subclasses of `Expression` to evaluate different kinds of expression, and when `ArithmeticExpression.evaluate()` calls the `evaluate()` method of its two operands, that method call could be despatched to any one of a hundred different implementations of the `evaluate()` method. In that situation, no inlining is possible, except perhaps in the case where one kind of operand (perhaps a numeric literal) is much more common than any other. By contrast, we're generating bytecode for a specific arithmetic expression where we know that the two operands are a literal and an atomizer, and in that situation inlining is eminently possible.

So the key difference is: in the interpreter, one Java method is handling all arithmetic expressions. In the generated bytecode, there's one Java method for each individual arithmetic expression in the stylesheet (provided of course that it's executed often enough to justify the code generation). An individual arithmetic expression knows statically what kinds of operands are; the generic code that handles all arithmetic expressions only finds this out when it gets executed.

- Avoiding boxing and unboxing. One of the consequences of using highly polymorphic methods like `Expression.evaluate()` is that data has to be passed from caller to callee, and back, in a form that satisfies a strongly typed interface. For example, the result of every XPath function call has to be returned as an instance of the class `net.sf.saxon.om.Sequence`. So with an XPath expression like `count($x) + 1`, the chances are that the implementation of `count()` is computing an integer, which has to be wrapped as an `net.sf.saxon.om.Sequence`, merely so that this can be unwrapped again in order to add one to the value. The bytecode generator is able to avoid a lot of this boxing and unboxing.

  How much does this matter? We don't really know. We know that the costs of allocating and garbage collecting short-lived objects are much less than they were in Java's early days, but small costs incurred millions of times do add up.

4. Elaboration

In this section we'll first look at requirements: what are we trying to achieve? Then we'll explain the concept of expression elaboration; and we'll illustrate it with an example.

4.1. Why try something new?

For this project we wanted to try a new technique, called expression elaboration, which I will go on to explain in the next section. But before doing so, I should explain why we were motivated to experiment with new ideas.

The immediate driver was the development of a new product (SaxonCS) targeted at the .NET Core platform. [1] For many years, we (Saxonica) delivered a version of Saxon for the .NET Framework platform, which was built by using the open source IKVM tool to convert the compiled SaxonJ JAR file into a .NET executable, and adding an API layer to integrate it with other facilities of the platform. In 2019, Microsoft announced that they planned to discontinue development of .NET Framework, and concentrate future work on .NET Core. Although the two platforms offer very similar capabilities at the API level, the internal engineering is very different, sufficiently so that IKVM would need a complete rewrite to make it work with .NET Core; which was unlikely to be forthcoming since the main developer of IKVM, Jeroen Frijters, announced that he had no enthusiasm to take the task on. As a result we needed to find a different way of bridging Saxon from the Java platform to .NET, and we did this by writing our own source code transpiler [XML London 2021]. With IKVM (perhaps surprisingly) our Saxon bytecode generation logic worked seamlessly on .NET — as soon as we generate bytecode, IKVM translates it on the fly to .NET 's equivalent. In the new transpiler-based product, this wasn't going to work.

For the Java platform, we're a little disillusioned with bytecode generation anyway, because there's a lot of code to maintain and the benefits, as we've seen, are quite modest. We wanted to see if there might be another way of getting the benefits with lower maintenance cost. Because of our business model where we offer a free open-source product alongside a commercial Enterprise Edition, it's useful to offer features like bytecode generation that provide an easy-to-understand turbo-charger to the base product. So we were reluctant to drop it entirely, but at the same time we wanted to see if we could do better.

On the Javascript product, SaxonJS, which is developed using completely separate source code, we had seen outstanding performance benefits from a technique we called expression elaboration. In fact, the benefits were so clearly apparent to the naked eye that we never took the trouble to make detailed measurements of the actual speed-up. We knew that we were unlikely to achieve the same kind of benefit with the Java product because we were starting with something that was already much more highly tuned; but it looked as if it might give us an alternative to bytecode generation for the SaxonCS version, and perhaps even enable us to drop bytecode generation from SaxonJ.

[1] The terminology has evolved. SaxonCS = Saxon on the .NET platform (primarily for C#); SaxonJ = Saxon on the Java platform; SaxonJS = Saxon on Javascript platforms (Node.js and browsers)

4.2. Expression Elaboration Explained

Expression elaboration starts with exactly the same expression tree that we use for interpretation, but it then splits the work of evaluation into two phases:

- The first time any expression node on the tree is evaluated, we construct a lambda expression, which we then leave on the tree for subsequent use. The name "elaboration" refers to this stage of the process.
- All subsequent evaluations of the expression then merely call this lambda expression, passing the evaluation context as an argument.

  That's a convenient way to explain it, but in practice when an expression is elaborated, this usually involves elaborating its subexpressions, and so on down to the bottom of the tree. So typically, the first time a user-written function or template is called, the body of the function is elaborated into a lambda expression, which invokes further lambda expressions held in its closure, and so on recursively; in the typical case the original expression tree then plays no further part.

Lambda expressions have become ubiquitous in nearly all modern programming languages, and the syntax and semantics are similar across Java, C#, and Javascript.

4.3. A simple example

Let's look at one particular instruction, called "negate". This implements the unary minus operator: it corresponds to an XPath expression such as `-$x`. [2] In SaxonJ, the code to evaluate a negate instruction in the interpreter looks like this:

```
@Override
public NumericValue evaluateItem(XPathContext context) throws
XPathException {
    NumericValue v1 = (NumericValue)
getBaseExpression().evaluateItem(context);
    if (v1 == null) {
        return backwardsCompatible ? DoubleValue.NaN : null;
    }
    return v1.negate();
}
```

Some observations:

[2] According to the XPath grammar, `-1` is a negate expression applied to a literal; but we sort that out during static analysis, so this will always appear as a constant at run-time. Unary minus operators are rarely used with operands other than numeric literals, but we've chosen them as our example because they are so simple.

- The method `evaluateItem()` takes the XPathContext as a parameter. There's a lot of information in this object, but the only thing we do is pass it on when evaluating the single operand (accessed as `getBaseExpression()`)
- The logic essentially does four things:
  - Evaluate the operand.
  - Cast the result to a `NumericValue` (we know this cast is safe, because static type analysis will have generated a guard expression on the expression tree to check or convert the value in cases where it is necessary).
  - if the value of the operand is null (representing an empty sequence) return either `NaN` or `null` depending on whether XPath 1.0 backwards compatibility is in force
  - call the `negate()` method on the `NumericValue`.

Now see what happens when we elaborate this instruction:

```
@Override
public ItemEvaluator elaborateForItem() {
    final NegateExpression exp = (NegateExpression)getExpression();
    final ItemEvaluator argEval =
makeElaborator(exp.getBaseExpression()).elaborateForItem();
    final boolean maybeEmpty =
Cardinality.allowsZero(exp.getBaseExpression().getCardinality());
    final boolean backwardsCompatible = exp.isBackwardsCompatible();
    if (maybeEmpty) {
        if (backwardsCompatible) {
            return context -> {
                NumericValue v1 = (NumericValue) argEval.eval(context);
                return v1 == null ? DoubleValue.NaN : v1.negate();
            };
        } else {
            return context -> {
                NumericValue v1 = (NumericValue) argEval.eval(context);
                return v1 == null ? null : v1.negate();
            };
        }
    } else {
        return context -> ((NumericValue)
argEval.eval(context)).negate();
    }
}
```

What's going on here? Remember that the method elaborateForItem() is called the first time a particular negate instruction is evaluated. It does the following:

- Gets the operand expression in the expression tree (`getBaseExpression()`)
- Elaborates the operand expression, returning a lambda function

- Examines the expression tree to see (a) whether the result of the operand may be an empty sequence, and (b) whether evaluation is in XPath 1.0 backwards compatibility mode
- Returns one of three different lambda functions, depending on these input conditions. The resulting function performs no run-time check for backwards compatibility, and no check for the operand being null unless this is actually a known possibility.

  If you're not familiar with lambda expressions in Java, there are three in this sample, all taking the form `params -> (expression | "{" statements "}")`. This corresponds to the lambda calculus notation `λ params : expr`, but neither Java nor any other of the mainstream programming languages was prepared to take the plunge of using Greek letters in the concrete syntax. The syntax denotes an anonymous function that takes a `context` object as its argument, and returns the result of evaluating the supplied expression (or statements) which typically depend both on the explicit `context` argument supplied by the caller, and on variables (such as `argEval`) that are in scope at the point where the lambda expression appears: the values of these variables are carried along with the function itself and are referred to as the function's *closure*.

So comparing the interpreted code with the generated lambda function, what have we achieved?

- We've eliminated the code that navigates the expression tree at run-time to locate the operand expression. Instead, the elaborated operand expression is present in the closure of the generated function, as variable `argEval`.
- We don't check at run-time for null values unless they can actually occur.
- The run-time logic doesn't need to consider whether backwards compatibility is in force or not: this decision has been "baked in".
- This is only saving us a few instructions; but negating a number is only one instruction, so in relative terms, we've cut out a lot of overhead.

I'm not going to show the code for bytecode generation of this expression, but I'll show what the generated bytecode looks like (with added comments for explanation). This bytecode is produced when compiling the XQuery function `declare function f:negate($x as xs:double) as xs:double {-$x};`

```
// load the first argument (the XPathContext)
ALOAD 1
// Get the stack frame holding local variables
INVOKEINTERFACE net/sf/saxon/expr/XPathContext.getStackFrame ();
INVOKEVIRTUAL net/sf/saxon/expr/StackFrame.getStackFrameValues ();
// Load the value of the variable at slot 0 on the stack frame
ICONST_0
```

```
    AALOAD
    // The value is in general a Sequence; call head() to get its first
and only item
    INVOKEINTERFACE net/sf/saxon/om/Sequence.head ();
    // Cast this to type NumericValue
    CHECKCAST net/sf/saxon/value/NumericValue
    // Invoke NumericValue.negate()
    INVOKEVIRTUAL net/sf/saxon/value/NumericValue.negate ();
    // Wrap the result in a SingletonIterator
    INVOKESTATIC net/sf/saxon/tree/iter/SingletonIterator.makeIterator
(Lnet/sf/saxon/om/Item;);
    // Return the iterator as the result of the XQuery function
    ARETURN
```

The only really signficant difference from the elaboration case is that bytecode for the operand expression is generated inline, rather than being invoked separately.

All three approaches (interpreter, compiler, elaborator) end up calling the library routine `NumericValue.negate()` to do the real work. This is a polymorphic method with different implementations for integers, decimals, double, and floats. In all three cases the JVM hotspot optimizer has the opportunity to optimize the call by inlining, but it's only likely to do so in practice if one of these types occurs much more frequently than the others.

It's possible that as a result of Saxon's static analysis the elaborator already knows what the type of the numeric value will be. With bytecode generation, we can easily pass this information to the Java compiler by casting to the relevant type instead of to the generic type NumericValue (though in fact, we fail to take advantage of this opportunity). With the interpreter, this isn't possible, because a single method is handling all Negate expressions in the query or stylesheet, and they will typically be handling different types of operand. For the elaborated case, we could do it in principle, by generating different lambda functions for the four cases, plus one for the case where the type is statically unknown. However, the complexity multiplies exponentially -- instead of generating one of three possible lambda functions, we would be generating one of 15, and it would need strong justification to attempt this.

4.4. Push mode, Pull mode

In the example above, the interpreter used a method `evaluateItem()` to evaluate the `negate` expression (and its operand). We use that method where the expression result will always be a singleton item (or perhaps an empty sequence). Other methods are used where an expression can return an arbitrary sequence. Elaboration, similarly, can generate code that uses different modes of evaluation.

At the top level, we have two ways of evaluating an expression: pull mode and push mode.

- In pull mode, the `iterate` method returns the result of the expression (which in the general case is a sequence) as an iterator over the items in the sequence. This means we are doing lazy evaluation, which is an important technique in all functional languages — it means that in many cases, evaluation of an expression can finish before the operands are fully evaluated, because enough information is available to establish the result.

  The `evaluateItem()` method seen in the example above is a short-cut method provided for convenience when an expression always returns a singleton result.
- In push mode, the results of the expression are not returned to the caller, but are written to a result stream, which will often be the final serialized result of a transformation. The advantage of push mode is that there is no need to hold the entire result document in memory, it can be written out "on the fly".

Both modes are supported by elaboration: for any given expression on the tree, we can generate either a pull function, or a push function, or both.

For pull mode, the function that we generate takes a single argument, the object holding the dynamic context, and it returns an iterator over the expression results. For push mode, we generate a function that takes two arguments, the dynamic context object and the destination to be written to; the function returns no result, but instead has a side-effect of writing to the destination.

Most instructions only support one mode of execution directly: for example an element constructor supports push mode, while an arithmetic expression or path expression supports pull mode. If the opposite mode is needed, it can be easily achieved using a wrapping function that converts the results. But some instructions - notably "flow of control" instructions such as conditional expressions, iteration instructions (xsl:for-each in XSLT, for expressions in XPath/XQuery), and function calls, support both modes natively.

5. Results

So, what benefits are we seeing from expression elaboration?

The results given here are provisional, for two reasons: firstly, implementation is incomplete (we've only implemented elaboration in SaxonJ for a selection of commonly used expressions), and secondly, measuring the effect is not easy.

At this point I need to acknowledge the contribution of Chris Newland, who has been working with us to improve our ability to benchmark the Saxon software and assess the impact of changes. Benchmarking Java applications is a very skilled task, and it's very easy to come to incorrect conclusions if you cut corners. Getting repeatable results (where today's figures come out the same as yesterday's) is challenging: don't try it on your laptop, where temperature variations or a low battery can cause the CPU speed to be throttled. Getting good reliable data needs a controlled stable machine configuration, and benchmark runs that take

hours rather than minutes. And even where the results are consistent, that's no guarantee that you will draw the right conclusions from the data.

We've been putting a lot of work into measurement on the Java platform, but I mentioned that part of the motivation was to see what we could achieve on .NET, where bytecode generation isn't an option. Our benchmarking activities on .NET have been far less thorough, but the early indications are very positive: for example the XMark query Q11 came down from 1192ms with the interpreter to 858ms with elaboration — a 28% improvement, better than we get with bytecode generation on the Java product. Both figures have an error margin of around ±5%. Other queries also showed a benefit, though not usually as great as this: 10% is more typical. With improvements of this order, we can probably declare victory and dispense with the effort of doing more accurate measurement.

On Java, so far, we're seeing much smaller improvements. For XMark Q11, for example, elaboration brings the timing down from 107ms to 104ms. Other queries show similar results: the improvement, if it exists, is hardly measurable, and is certainly a lot less than we get with bytecode generation. Needless to say, this is disappointing.

Of course, there is a beneficial side-effect: when you put this much effort into instrumentation, you discover all sorts of opportunities for performance improvements that you weren't actually looking for, and following up on some of these opportunities has probably distracted us from the task we set out to accomplish. But they're out of scope for this paper.

We're still exploring why the benefit on Java is so small, and whether there's anything we can do to improve matters. We've found that some of the switches that Java provides to control the behaviour of the hot-spot compiler can make a significant difference, but in the real world that's not very useful knowledge since very few Saxon users in the field are likely to take advantage of it. And many of the Saxon users who do care deeply about performance probably have applications in which Saxon is just one of many components. But observing how these switches affect the results does give us clues about how the lambda functions we're generating are treated by the hot-spot optimizer.

Perhaps the key finding (though a provisional one that we need to confirm) is that the hot-spot optimizer is taking no notice of the values in the closure of a lambda expression: just because a boolean variable in the closure is false, doesn't mean that the hot-spot compiler is eliminating a run-time code branch that depends on that value. That's because it's not optimizing for a particular expression in the query or stylesheet (say, a particular filter predicate), rather it's optimising for a statistical average of all filter predicates in the stylesheet. The net result is that with both elaboration and interpretation, the ability of the hot-spot optimizer to work its magic is inhibited by the fact that the calls we are making to evaluate subexpressions are so heavily polymorphic.

It seems fairly clear that there's some significant difference in the way the Java and C# optimizers handle lambda expressions that cause the technique to show greater benefits in C# than on Java. But so far, we haven't been able to pin down exactly what it is.

A Benchmark Collection of Deterministic Automata for XPath Queries

Antonio al Serhali
Inria

Joachim Niehren
Inria

Abstract

We provide a benchmark collection of deterministic automata for regular XPath queries. For this, we select the subcollection of forward navigational XPath queries from a corpus that Lick and Schmitz extracted from real-world XSLT and XQuery programs, compile them to stepwise hedge automata (SHAs), and determinize them. Large blowups by automata determinization are avoided by using schema-based determinization. The schema captures the XML data model and the fact that any answer of a path query must return a single node. Our collection also provides deterministic nested word automata that we obtain by compilation from deterministic SHAs.

1. Introduction

XML is one of the most used standardized formats for representing exchanging structured data between various tools and applications. XML documents form unranked data trees. Processing XML documents in both in-memory and streaming modes are widely studied for many years [24] [25] [28] [27] [26]. The most frequent tasks are validating, querying and transforming XML documents. In the XML technology, this is done with standardized languages based on XPath queries, such as XSLT and XQuery.

Automata-based algorithms are not only relevant for validating XML documents with respect to a schema (as with RelaxNG) but also for querying XML streams [7] [6] [16] [11]. The problem with syntax-oriented approaches for answering XPath queries on XML streams yield only low coverage. Automata approaches, in contrast, can deal with all of XPath3.0 as shown by Sebastian, Niehren, and Debarbieux [6]. When applying automata, however, it is natural to abstract XML documents to nested words [23], which generalize on unranked data trees and sequences thereof that are also called forests or hedges. Automata

for nested words are also relevant for enumerating query answers of document spanners in in-memory mode [8] [20], and for enumerating query answers on data trees [14] [2] [5].

Deterministic automata are relevant to keep the computational complexity of various problems tractable. In particular it enables automata minimization in polynomial time and universality testing in linear time. In contrast, universality becoming EXP-complete for nondeterministic automata on trees or nested words [4] [21]. Note that universality testing can be used as a stopping condition for automata algorithms. More concretely, determinism is required for the streaming algorithms of [7] and [11] but also for the inmemory algorithm of [20]. Therefore, deterministic automata on nested words need to be produced for regular path queries [12] [15] [9] for benchmarking these algorithms.

Compiling regular path queries to automata is less problematic, but their determinization may blowup the automata sizes exponentially. This also happens in practice. For the XPath query `//a[following- sibling::b[.//c][./d]]/e` for instance, [6] construct a nested word automaton (NWA) with 38 states of overall size 7338. The determinization of this automaton has more than 5000 states and 20 million transition rules. It is so big that it cannot be computed an a standard laptop. This shows that the usual determinization algorithm for NWAs [3] [1] [19] quickly leads to a size explosion. Niehren and Sakho [17] improved this situation by using the determinization algorithm for stepwise hedge automata (SHAs), which in turn can be compiled to deterministic NWAs. In this way deterministic SHAs and NWAs of decent size could be obtained for the 10 forward navigational XPath queries for the XPathMark benchmark [10]. But even the determinization of SHAs may lead to unreasonably large automata for pratically relevant XPath queries. For the XPath query `/a/b//(* | @* | comment() | text())`, for instance, a deterministic SHA with 145 states and size 348 got reported, whose determinization has 10 005 states and overall size 1 634 123 [18].

Niehren, Sakho, and Al Serhali showed recently [18] that this determinization problem for SHAs can be solved by using schemas, i.e., deterministic automata that model which nested words are valid inputs of the automaton. In the case of XPath queries, the schema captures the XML data model, and that each query answer must return a unique node of the XML document.

The first schema-based approach is to determinize the product of the query automaton with the schema automaton. For the above XPath query, this yields a deterministic SHA with 92 states and size 417, which after minimization goes down to 27 states and size 98. Nevertheless, this approach may seem surprising at first sight, since the schema-product is usually bigger than the query automaton itself. But indeed it works quite nicely in practice. The intuition is that the deterministic schema reduces the number of subsets of states that are to be considered during determinization since all states in such subsets must be aligned to the same schema state.

The second schema-based approach is to clean the determinized automaton with respect to the schema. This means removing all states and transitions that cannot be aligned to the schema. Schema-based cleaning has the advantage of always yielding smaller automata. Unfortunately, however, it is not always computationally feasible in practice, since the automaton produced by determinization is often too large for being schema-cleaned.

The third schema-based approach is schema-based determinization, an algorithm proposed in [18]. The idea is integrate schema-based cleaning directly into the determinization algorithm, in order to avoid large blowups from the beginning, while producing the same result as with the second approach. The automata obtained by schema-based determinization are usually smaller than by determinizing the schema-product, also after minimization, since they do not recognize the same language.

We applied the implementations of all three approaches to show that small deterministic SHAs and NWAs can be obtained for all the regular XPath queries in the benchmark corpus that Lick and Schmitz [30] [13] harvested from XSLT and XQuery programs available online (docbook, teixml, htmlbook, ...). The third solution based on schema-based determinization followed by minimization yields the best results. The largest SHA obtained in this way for the whole benchmark collection has 58 states. In average there are 22 states and 71 transition rules per automaton. All automata are published in the software heritage archive at https:// archive.softwareheritage.org/ browse/ origin/ ?origin_url=https:// gitlab.inria.fr/aalserha/xpath-benchmark.

The fact that we can indeed determinize the automata of most if not all practical XPath queries with a mild size increase, gives new hope to improve the situation on XML streaming in the near future, building on approaches requiring deterministic automata [7] [11] [29].

1.1. Outline

We present our selection of regular XPath benchmark queries from the corpus of Lick and Schmitz [30] in Section 2. Nested words and their relationship to XML documents are recalled in Section 3. A deterministic stepwise hedge automata defining the schema of valid XML documents is given in Section 3.2. A formal definition of stepwise hedge automata follows for the sake of self-containedness in Section 4. In Section 5 we discuss our compiler from XPath expressions to deterministic automata, and illustrate it by example automata from our benchmark collection. In Section 6 we discuss how we tested our automata for correctness on a sample of annotated XML documents produced from the XPath query based on Saxon XSLT. The sizes of automata in our benchmark collection of SHAs are discussed in Section 7. We conclude in Section 8. Some complementary information can be found in Appendix A.

2. XPath Benchmark Queries

We start with the collection of 21000 XPath queries that Lick and Schmitz [30] extracted from real-world XQuery and XSLT programs available on the Web. The purpose of this corpus is to reflect the form and distribution of XPath queries in practical applications. The much smaller XPathMark benchmark [10], in contrast, focuses on functionality testing.

We then filter the subclass of around 4500 forward navigational XPath queries of Lick's and Schmitz's corpus. The other queries contain comparisons of data values, arithmetics, and functions, including higher-order functions to iterate over sequences, which may be nonregular. We also removed boolean queries and kept only node selection queries. We then selected the 180 largest queries of this subcorpus.

Finally, we removed duplicates of queries up to renaming of XML namespace prefixes and local names, and syntactical details, such as `.// author` or `descendant-or-self::author` or `descendant-or-self::corpauthor`. This leads us to the collection of 79 queries. The first 10 queries are shown in Table 1.

Table 1. The first 10 of the 79 queries of the benchmark collection (see Table A.1)

Id	XPath Query
18330	`/descendant-or-self::node()/child::parts-of-speech`
17914	`/ descendant-or-self::node()/ child::tei:back/ descendant-or-self::node()/child::tei:interpGrp`
10745	`*//tei:imprint/tei:date[@type='access']`
02091	`* | .//reftd`
00744	`.//@id | .//@xml:id`
12060	`.//attDef`
02762	`.//authorgroup/author | .//author`
06027	`.//authorinitials | .//author`
02909	`.//bibliomisc[@role='serie']`
06415	`.//email | address/otheraddr/ulink`

We note that the XPath query 18339 is considered as large since it contains the recursive axis `descendant-or-self`. Other queries are considered as large since having a parse tree with more than 15 nodes, for instance 05684 and 05684.

3. Nested Words for XML Documents

We use nested words to abstract from XML documents since automata can be defined more easily for nested words.

3.1. Nested Words

Nested words generalize on words by adding parenthesis that must be well-nested. Nested words also generalize on unranked trees and over sequences thereof that are often called hedges. We restrict ourselves to nested words with a single pair of opening and closing parenthesis ◎ and ◎ since named parenthesis can be encoded easily. Let Σ be a set that we call the alphabet. Nested words in $\mathcal{N}_\Sigma$ have the following abstract syntax.

$$w, w' \in \mathcal{N}_\Sigma \quad ::= \quad \varepsilon \mid a \mid \langle w \rangle \mid w \cdot w' \qquad \text{where } a \in \Sigma.$$

We assume that concatentation $\cdot$ is associative, and that the empty word ε is a neutral element, that is $w \cdot (w' \cdot w'') = (w \cdot w') \cdot w''$ and $\varepsilon \cdot w = w = w \cdot \varepsilon$. Nested words can be identified with hedges, i.e., sequences of unranked trees and letters, that is $\mathcal{N}_\Sigma = (\Sigma \cup \langle \mathcal{N}_\Sigma \rangle)^*$.

3.2. XML Documents

XML documents are labeled unranked trees that can be serialized into a text, such as for instance:

```
<s:a name="uff"> <s:b> gaga <s:d/> <s:b/> <s:c/> <s:a>
```

We represent XML documents as nested words over the signature Σ_{XML} that contains 4 disjoint types of letters: the XML node-types $\{elem, attr, text, comment\}$, the XML namespaces of the document $\{s\}$, the XML names of the document $\{a, \ldots, d, name\}$, and the characters of the data values, say UTF8. For the above example, we get the nested word:

$$\langle elem \cdot s \cdot a \cdot \langle attr \cdot name \cdot u \cdot f \cdot f \rangle \langle elem \cdot s \cdot b \cdot \langle text \cdot g \cdot a \cdot g \cdot a \rangle \langle elem \cdot s \cdot d \rangle \rangle \langle elem \cdot s \cdot c \rangle \rangle$$

4. Automata for Nested Words

Stepwise hedge automata (SHAs) [17] extend on classical finite state automata (NFAs) from words to nested words. They provide a graphical way to define regular languages of nested words, and thus regular languages of XML documents. SHAs are often easier to read that the better-known nested word automata (NWAs) and help us to avoid large size blowups coming with NWA determinization. In this section we recall the definition of SHAs based on the definition of NFAs and discuss their relationship NWAs.

4.1. Finite State Automata (NFAs)

We consider finite state automata with else rules and possibly infinite alphabets.

Definition. An *NFA (with else rules)* is a tuple A= $(\Sigma, Q, \Delta, I, F)$ such that alphabet Σ is a possibly infinite set, $\Delta = \Delta' \uplus \_^{\Delta}$ contains a subset of transition rules for letters $\Delta' \subseteq (Q \times \Sigma) \times Q$ and a subset of else rules $\_^{\Delta} \subseteq Q \times Q$. We call NFA A *deterministic* or equivalently a DFA if Δ' and $\_^{\Delta}$ are partial function.
As usual when using automata, we draw NFAs as graphs whose nodes are the states. A state $q \in Q$ is drawn with a circle (q) , an initial state $q \in I$ with an incoming arrow $\to(q)$, and a final state with a double circle $((q))$. A letter transition rule $(q_1, a, q_2) \in \Delta'$ is drawn as a black edge $(q_1) \xrightarrow{a} (q_2)$ that is labeled by a letter $a \in \Sigma$. An else rule $(q, q') \in \_^{\Delta}$ is drawn as $(q) \Rightarrow (q')$. It permits that the automaton in state q can go to state q' when reading any letter $a \in \Sigma$ such that there exists no q'' with $q \xrightarrow{a} q'' \in \Delta$. Any else rule can be expanded to a set of letter transitions rules as follows:

$$\frac{q \Rightarrow q' \in \Delta \quad a \in \Sigma \quad \neg\exists q'' \in Q.\ q \xrightarrow{a} q'' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{exp}} \qquad \frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{exp}}$$

4.2. Stepwise Hedge Automata (SHAs)

We extend NFAs to SHAs by adding adding apply rules that read states of subtrees rather than letters from the alphabet.

Definition. An *SHA (with else rules)* is a tuple $A = (\Sigma, Q, \mathcal{P}, \Delta, I, F)$ where $\Delta = \Delta' \uplus \Delta''$ so that $A' = (\Sigma, Q, \Delta', I, F)$ is a NFA. Furthermore, $\mathcal{P}$ is a finite set of tree states and $\Delta'' = (\langle\rangle^{\Delta}, @^{\Delta}, \dashrightarrow^{\Delta})$ such that $\langle\rangle^{\Delta} \subseteq Q$ is a subset of tree initial states, $@^{\Delta} \subseteq (Q \times \mathcal{P}) \times Q$ a set of apply rules, and $\dashrightarrow^{\Delta} \subseteq Q \times \mathcal{P}$ a set of tree final rules.
We draw SHAs as graphs extending on the graphs of NFAs. A tree state $p \in \mathcal{P}$ is drawn in gray (p) . A tree initial state $q \in \langle\rangle^{\Delta}$ is a hedge state is drawn as $\xrightarrow{\langle\rangle}(q)$ with an incoming tree arrow. An apply rule $(q_1, p, q_2) \in @^{\Delta}$ is drawn by a blue edge $(q_1) \xrightarrow{p} (q_2)$ carrying a state $p \in \mathcal{P}$ rather than a letter $a \in \Sigma$. It states that a nested word in state $q_1 \in Q$ can be extended by a tree in state $p \in \mathcal{P}$ and go into state $q_2 \in Q$. A tree final rule $(q, p) \in \dashrightarrow^{\Delta}$ is drawn as $(q) \dashrightarrow (p)$. It states that if w is a nested word in state $q \in Q$ then $\langle w \rangle$ is a tree in state $p \in \mathcal{P}$.

Transitions of SHAs have the form $q \xrightarrow{w} q' \text{ wrt } \Delta$ where $w \in \mathcal{N}_{\Sigma}$ and $q, q' \in Q$. They are defined by the inference rules:

$$\frac{q \in Q}{q \xrightarrow{\varepsilon} q \text{ wrt } \Delta} \qquad \frac{q \xrightarrow{a} q' \in \Delta^{exp}}{q \xrightarrow{a} q' \text{ wrt } \Delta} \qquad \frac{q_0 \xrightarrow{w_1} q_1 \text{ wrt } \Delta \qquad q_1 \xrightarrow{w_2} q_2 \text{ wrt } \Delta}{q_0 \xrightarrow{w_1 \cdot w_2} q_2 \text{ wrt } \Delta}$$

$$\frac{q' \in \langle\rangle^{\Delta} \qquad q' \xrightarrow{w} q \text{ wrt } \Delta \qquad q \longrightarrow p \in \Delta \qquad q_1 \xrightarrow{p} q_2 \in \Delta}{q_1 \xrightarrow{\langle w \rangle} q_2 \text{ wrt } \Delta}$$

The last inference rule says that when reading a tree $\langle w \rangle$ the automaton can transit from a state q_1 to a state q_2 if with w it can transit from some tree initial state q' to q , so that there is some tree final rule $q \longrightarrow p \in \Delta$ and some apply rule $q_1 \xrightarrow{p} q_2 \in \Delta$. The language $\mathcal{L}(A)$ of a SHA is defined as usual for NFAs except that nested words may be recognized too:

$$\mathcal{L}(A) = \{w \in \mathcal{N}_\Sigma \mid q \xrightarrow{w} q' \text{ wrt } \Delta,\ q \in I,\ q' \in F\}$$

The notion of determinism for SHAs extends on the notion of left-to-right determinism of NFAs and on the notion of bottom-up determinism of tree automata.

Definition. We call an SHA A *deterministic* or equivalently a *dSHA*, if the contained finite automaton A' is a DFA, there is at most one tree initial state in $\langle\rangle^{\Delta}$, and $@^{\Delta}$ and $\longrightarrow^{\Delta}$ are partial functions.

4.3. Adding Typed Else Rules

Suppose that the alphabet Σ is typed, in that any letter $a \in \Sigma$ can be given some types in some type set T . We can then add typed else rules $(q, \tau, q') \in \Delta \times T \times \Delta$ that we draw as $\textcircled{q} \xrightarrow{-\tau} \textcircled{q'}$. In contrast to untyped else rules, a typed else rule cannot be expanded with all letters from Σ , but only with those that can be given the type τ .

4.4. A Schema for XML Documents

The most frequent type of XPath queries select nodes of XML documents. For referring to selected nodes, we fix a single selection variable $\mathtt{x}$. We call an XML document or subdocument, in which a single node is annotated by $\mathtt{x}$, an *x-annotated example*. An $\mathtt{x}$-annotated example is called *positive* for a query if the query selects the $\mathtt{x}$-annotated node in the XML document, and *negative* otherwise.

Figure 1 recognizes the set of all $\mathtt{x}$-annotated examples. These must satisfy the XML data model and contain exactly one occurence of $\mathtt{x}$.

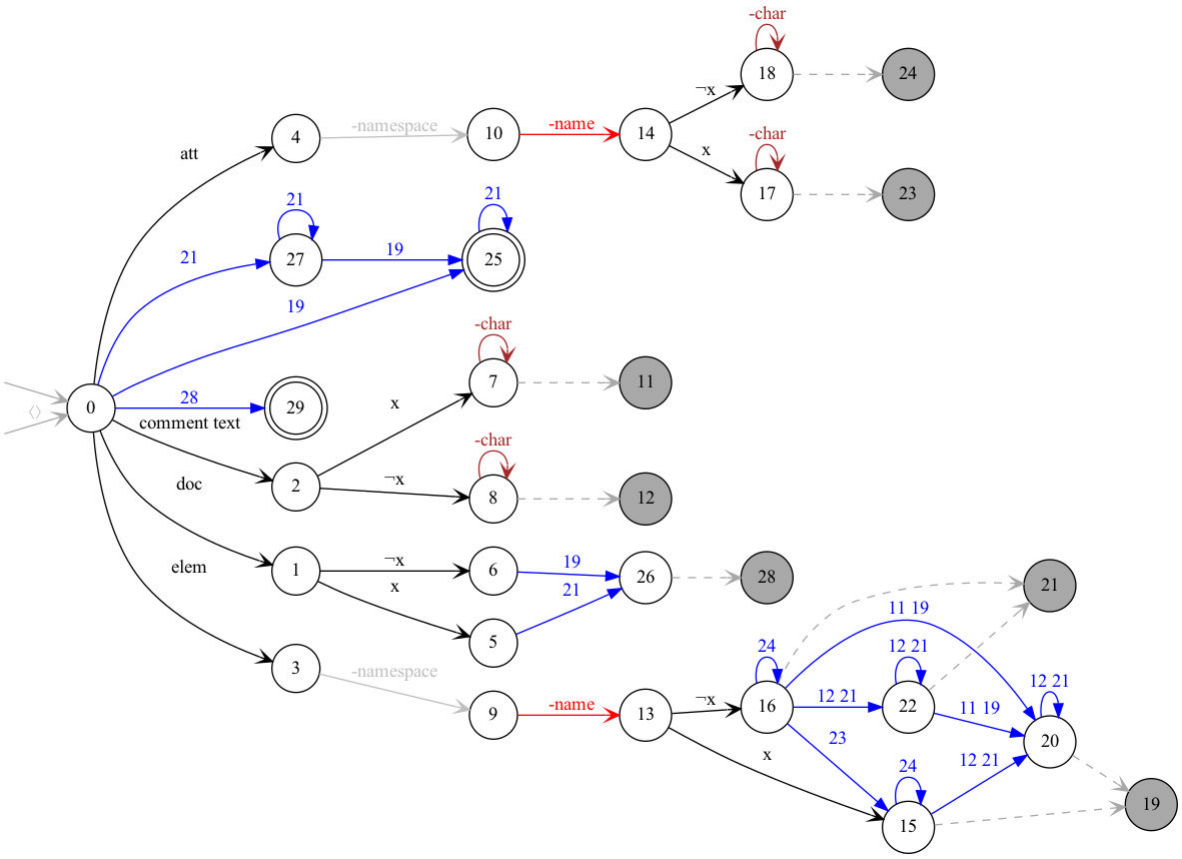


Figure 1. The dSHA $xml\&one^x$: a schema for x-annotated XML documents

The automaton starts in hedge state 0 where it expects to read a nested word $\langle w \rangle$, that can be evaluated to tree state `28`, in order to go to the final state `29`, where it accepts. The sequence of children w of the tree must be evaluated form the tree initial state, which is equally the hedge state `0`. If w starts with letter `doc` indicating an XML document node at the root, the automaton moves from state `0` to state `5`. There it may either read the variable `x` and go to state `5`, where it expects a subtree in state `21`, i.e. an XML element of which no node is annotated by `x`. Or it may read the symbol ¬ `x` and move to state `6`, where it expects a subtree in state `19`, i.e. an XML element of which exactely one node is annotated by `x`. In both cases it can go to the hedge state `26` and from there to the tree state `28`. The automaton also states the relationships of elements, attributes, text and comment nodes according to the XML data model.

The alphabets of names and namespaces of XML documents are infinite. In order to represent infinite sets of transition rule symbolically in a finite manner, the automaton use type else rules. The typed else rule in state `3`, for instance, is labeled by `-namespace`, permitting to read any namespace and to go to state `9`. State `9` in turn has an else rule labeled by `-name` which permits to read any (local) name and move to state `13`.

4.5. Nested Word Automata (NWAs)

Nested word automata (NWAs) [[19]][[1]] are well known pushdown machines for defining regular languages of nested words. They can process nested words in a streaming manner: top-down, left-to-right, and bottom-up manner. SHAs in contrast operate bottom-up and left-to right only. They avoid any top-down processing, since it quickly leads to huge size increases during NWA determinization.

Any SHA can be compiled in linear time to an NWA such that determinism is preserved. There also exists an inverse translation in quadratic time (but not preserving determinisim), so both automata classes have the same expressiveness, also when restricted by determinism. We omit the details, but provide deterministic NWAs in our collection. See for instance: Figure 4.

5. Compiler to Automata

We extended on the compilation chain for regular XPath queries to automata from [17]. As a running example, we consider the following query:

Q_2 : `h:body[@lang != '']`

Query Q_2 selects a node if it has a child named `body` in namespace `h`, that has the attribute node named `lang` containing a nonempty text.

5.1. Parser

Our parser for XPath expressions computes a parse tree following the grammar of XPath 3.1 from the W3C. In addition, it returns for any forward regular XPath expression a logical formula in the language FXP [6]. For the XPath example Q_2 , we obtain the following FXP formula:

$$child(lab_{elem:type} \wedge lab_{h:namespace} \wedge lab_{body:name} \wedge lab_{x:var} \wedge \\ child(lab_{att:type} \wedge cand_{default:namespace} \wedge lab_{lang:name} \wedge string \neq ''))$$

Our previous parser needed considerable improvement in order to be able to cover the large variety of queries from the corpus of Lick and Schmitz [30].

5.2. Nested Regular Expressions

We next compile formulas to nested regular expressions, which extend on standard regular expressions from words to nested words. Again, considerable work was needed to enable a sufficiently large coverage. For the query Q_2 our compiler yields the nested regular expression:

$\langle(elem{:}type._) + doc{:}type).\_.\top.$
$\langle elem{:}type.h{:}namespace.body{:}name.x{:}var.$
$\langle att.type.default{:}namespace.lang{:}name.\_.(\_char.(\_char)^*\rangle.\top\rangle.\top\rangle.\top$

Note that the test for a nonempty string got translated by the regular expression $\_char.(\_char)^*$. It should also be noticed that this expression matches some x-annotated nested words, that are not x-annotated examples, i.e. not belonging to the language $\mathcal{L}(xml\&one^x)$ of the schema. This is since the nested subwords matching universal expression $\top$ are completely unconstrained.

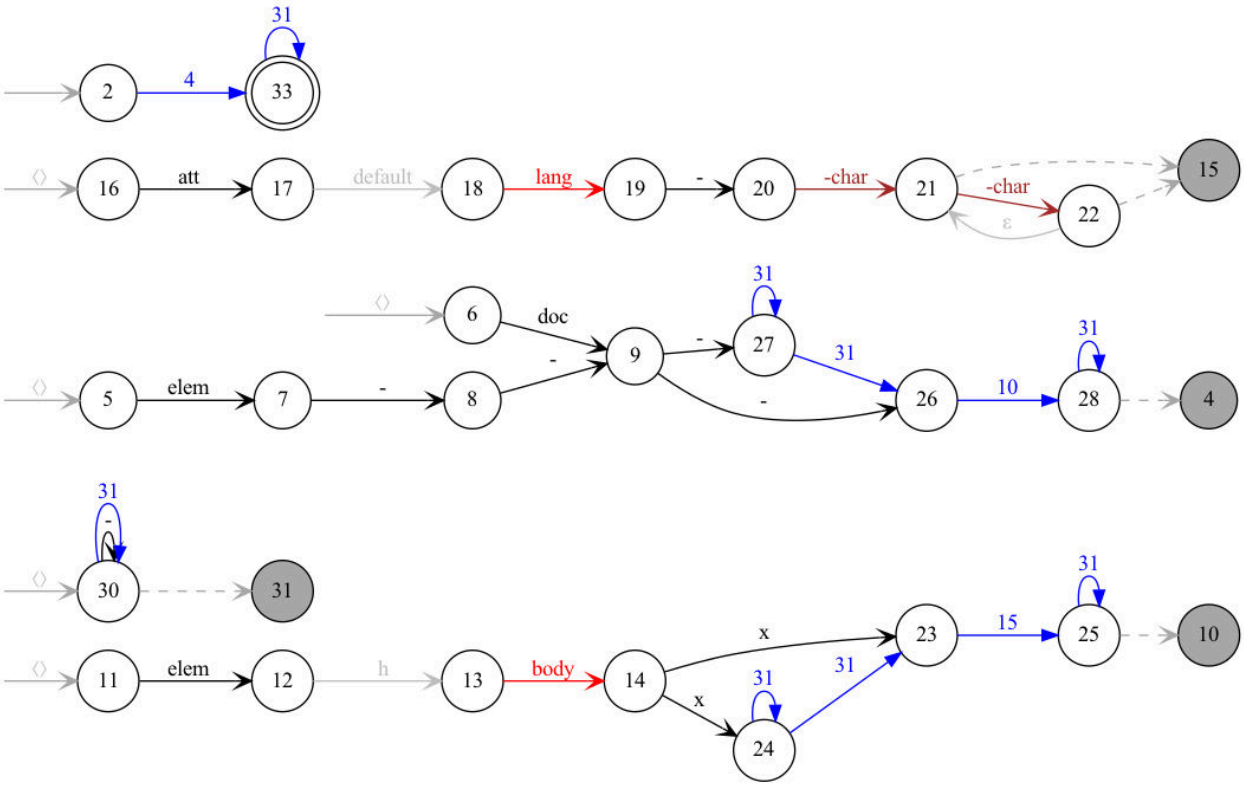


Figure 2. The nondeterministic SHA $A_2 = sha(Q_2)$ **.**

5.3. Compiler to SHAs

The compiler then converts nested regular expressions into SHAs. This is done by extending a usual compiler from regular expressions to NFAs. The interaction of recursion and nesting leads to some nasty issues, that are discussed and resolved in [17]. For developping the present benchmark, we needed to add a treatment of typed wildcards such as `-char`. This is done by introducing typed else rules. For the query Q_2 we obtain Figure 2. Similarly to the nested regular expression, this SHA may recognize some annotated nested words, that are not x-annotated examples, i.e., that do not belong to the language of the schema $L(xml\&one^x)$.

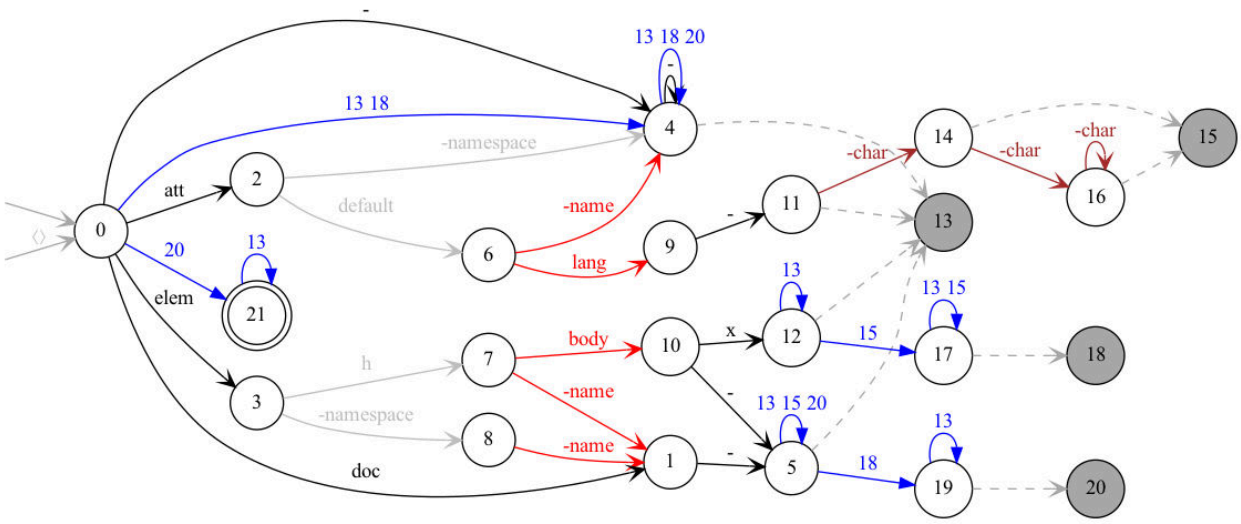


Figure 3. The schema-based determinization $det_S(A_2)$ where $S = xml\&one^x$

5.4. Determinization

The usual determinization algorithms for NFAs and tree automata can be lifted to a determinization algorithm for SHAs. When applied to query Q_2 however, we obtain a SHA with 25 states and 183 transition rules, which is much larger than one might expect. It is given in Figure A.1 of the appendix. Even worse, in some cases, the determinization algorithm does not finish after some hours.

5.5. Determinizing the Schema Product

Determinization applied to the product of the queries' automaton and the schema $xml\&one^x$ permits to compute deterministic automata for all queries of our benchmark within a timeout of 100 seconds. The result for Q_2 is a dSHA with 53 states and 110 transition rules, see automaton Figure A.2 of the appendix. The overall size is smaller, and the automaton is much easier to understand, but the number of states increased.

5.6. Schema-Based Determinization

Schema-based determinization as proposed in [18] improves the situation further. For query Q_2 it yields: SHA in Figure 3 which has only 22 states and 45 transitions. The size is roughly devided by 2 compared to: Figure A.2.

5.7. Minimization

We then minimize the dSHA from Figure 3. This often reduces the size and the number of states in an important manner and often makes it easy to see how the automaton is functioning. Exceptionally in the case of Q_2 , no states are fusioned when minimizing the dSHA obtained by schema-based determinization.

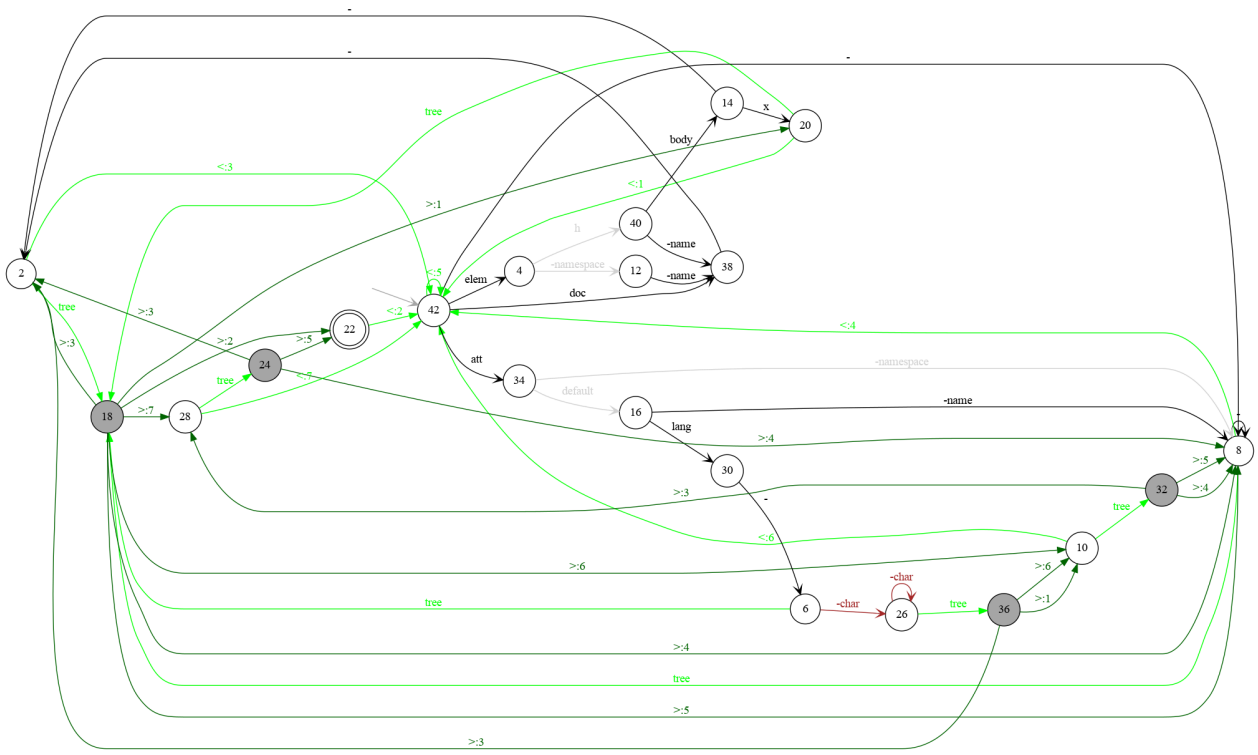


Figure 4. The deterministic NWA $nwa(det(A_2))$ obtained from the dSHA $det(A_2)$

It should be noticed that minimizing the determinization of the schema product usually yields a different result then minimizing the schema-based determinization. This is since both automata may recognize different languages. Some nested words outside the schema may be accepted after schema-based determinization, but not by the schema product.

5.8. Compiler to NWAs

The compiler finally maps SHAs to in linear time, while preserving determinism. For instance the minimal dSHA in Figure 3 is converted to: Figure 4.

6. Testing Automata on Samples

For testing the stepwise hedge automata, we created a sample with positive and negative x-annotated examples for each of the queries. Please contact the authors if you are interested in the test samples. They can be provided without problem.

For this we produced an XML document for each of the XSLT programs from which the XPath queries of Lick and Schmitz were extracted. We did this in such a way that each of the queries has at least one answer on one of the subdocuments of the document of its collection. Subdocuments are important here, since the XPath queries of an XSLT program will be applied to subdocumens naturally.

By using Saxon XSLT, we computed the answer set of all the queries on all the subdocument of the produced XML documents. For this, we exported query

answers in Dewey notation, similarly to the way that nodes are returned by Schematron: The Dewey noation of a node is its relative address from the root, i.e., by the list of child steps leading to the node. Such lists can be easily encoded in XML format.

Each query query answer yields a positive x-annotated examples for the query, that is obtained by annotating the XML document by x at the selected position. Negative x-annotated examples are obtained from the answers of the other queries on the same document. The annotation of the XML document is done by yet another XSLT stylesheet that we wrote for this purpose. Here we use the fact that query answers are also represented in XML format.

By testing the automaton on these samples, we could fix various problems that arised on the way to our final collection. Currently, no test failures are remaining, except for the query 13896 below that we removed from the corpus for the current version. The problem here is raised by the blank symbol in the attribute value `'evans citation'`:

```
//HEADER//IDNO[@TYPE='evans citation']
```

7. Statistics of the Benchmark Automata

We compiled all of our 79 XPath queries to deterministic automata using the compilation chain described in Section 5. Here we present the statistics of the benchmark automata that we obtained. The summary is given in Table 2. We show for each automaton two numbers `size(#states)` where `size` is the overall size of the automaton and `#states` the number of its states.

The nondetermnistic SHAs compiled from the nested regular expressions was cleaned using the schema $xml\&one^x$: Figure 1. The result is called $A = sha(Q)$ leading to the statistics in the second column of Table 2.

We note that 37% of the SHAs original stepwise hedge automata for the queries $A = sha(Q)$ have more than 100 states, so they are sometimes bigger than one might expect. The biggest is for query 06176 with 630 states and an overall size of 1391. The reason is that this query is selecting a union of 20 subqueries, all with descendant-or-self axis. Foe each subquery, we have 4 construsts of respective state sizes: 2, 6, 10 and 13, making a subtotal of $31 * 20 = 620$. With an additional 8 states for one subquery that select all descendants with an attribute named id and another 2 for reading any tree, we end up with our total 630 states.

Table 2. Statistics on the automata for the XPath queries from Table 2

$Query\ Q$ id	$A = sha(Q)$	$det(A)$	$B = det(A \times S)$	$C = det_S(A)$	$B' = mini(B)$	$C' = mini(C)$	$nwa(C')$
18330	99 (41)	465 (43)	145 (44)	74 (22)	128 (39)	61 (18)	73 (18)

$Query\ Q$ id	$A = sha(Q)$	$det(A)$	$B = det(A \times S)$	$C = det_S(A)$	$B' = mini(B)$	$C' = mini(C)$	$nwa(C')$
17914	179 (75)	2740 (141)	265 (69)	150 (44)	152 (43)	82 (24)	98 (24)
10745	187 (76)	939 (68)	275 (72)	141 (38)	218 (57)	130 (34)	150 (34)
02091	100 (42)	555 (45)	182 (57)	81 (24)	146 (44)	61 (17)	75 (17)
00744	109 (46)	335 (37)	169 (54)	80 (24)	128 (41)	54 (15)	64 (15)
12060	64 (25)	162 (22)	139 (44)	56 (16)	121 (39)	44 (12)	54 (12)
02762	121 (50)	564 (53)	222 (63)	97 (28)	123 (39)	46 (12)	56 (12)
06027	115 (48)	1101 (79)	184 (57)	82 (24)	123 (39)	46 (12)	56 (12)
02909	96 (38)	311 (36)	213 (62)	100 (27)	167 (49)	91 (24)	105 (24)
06415	139 (58)	1793 (93)	300 (74)	135 (36)	229 (55)	101 (25)	123 (25)
03257	130 (53)	1310 (92)	445 (85)	224 (46)	210 (49)	87 (20)	105 (20)
05122	83 (33)	292 (33)	221 (55)	92 (23)	161 (44)	63 (16)	77 (16)
09138	269 (117)		323 (97)	164 (49)	133 (40)	56 (13)	66 (13)
05460	232 (98)	3468 (174)	509 (127)	269 (77)	156 (44)	62 (16)	76 (16)
12404	84 (33)	258 (31)	170 (52)	77 (22)	143 (44)	68 (19)	82 (19)
10337	92 (36)	291 (34)	197 (58)	92 (25)	159 (47)	83 (22)	97 (22)
06639	123 (50)	516 (49)	237 (65)	106 (30)	154 (44)	60 (16)	74 (16)
14340	79 (33)	231 (29)	126 (40)	58 (18)	110 (36)	45 (14)	55 (14)
13804	70 (29)	155 (21)	128 (41)	63 (20)	124 (40)	60 (19)	70 (19)
02194	81 (33)	253 (31)	135 (42)	66 (20)	119 (38)	53 (16)	63 (16)
06726	149 (64)	2806 (149)	176 (53)	97 (30)	121 (38)	55 (16)	65 (16)
13640	100 (41)	364 (40)	165 (50)	86 (26)	140 (43)	76 (23)	90 (23)
05735	111 (45)	412 (44)	201 (58)	106 (30)	161 (47)	96 (27)	110 (27)
15766	144 (58)	669 (60)	300 (77)	155 (41)	219 (57)	135 (35)	151 (35)
15539	217 (88)	1709 (121)	402 (98)	213 (58)	228 (57)	144 (38)	164 (38)
15809	197 (84)	3795 (188)	230 (67)	129 (39)	145 (43)	82 (24)	96 (24)
15524	125 (50)	471 (49)	245 (68)	130 (35)	185 (52)	120 (32)	134 (32)
06512	135 (56)	583 (58)	218 (60)	117 (35)	152 (43)	77 (23)	91 (23)
06176	1391 (630)		1661 (448)	1203 (386)	176 (43)	113 (23)	127 (23)
12539	179 (76)	3479 (174)	243 (69)	138 (40)	166 (48)	101 (28)	115 (28)
11780	205 (88)	3832 (190)	254 (71)	143 (41)	164 (47)	99 (27)	113 (27)

$Query\ Q$ id	$A = sha(Q)$	$det(A)$	$B = det(A \times S)$	$C = det_S(A)$	$B' = mini(B)$	$C' = mini(C)$	$nwa(C')$
11478	101 (41)	365 (40)	166 (50)	87 (26)	141 (43)	77 (23)	91 (23)
11227	153 (62)	583 (53)	334 (81)	163 (42)	244 (59)	144 (37)	166 (37)
05684	1348 (616)		1068 (284)	719 (226)	193 (39)	124 (16)	134 (16)
06947	744 (342)		828 (232)	444 (129)	151 (41)	71 (14)	83 (14)
06794	270 (121)		354 (102)	178 (51)	144 (42)	64 (15)	76 (15)
06169	346 (155)		427 (121)	219 (62)	147 (41)	67 (14)	79 (14)
06924	598 (274)		682 (192)	362 (105)	147 (41)	67 (14)	79 (14)
11958	109 (44)	348 (35)	213 (57)	90 (24)	178 (48)	76 (20)	94 (20)
01705	772 (350)		1308 (279)	746 (172)	221 (48)	113 (19)	129 (19)
02086	809 (367)		1366 (291)	781 (180)	223 (48)	115 (19)	131 (19)
02000	642 (291)		723 (201)	387 (110)	163 (41)	83 (14)	95 (14)
02697	383 (172)		464 (131)	240 (68)	149 (41)	69 (14)	81 (14)
14183	110 (44)	362 (36)	217 (58)	94 (25)	182 (49)	80 (21)	98 (21)
07106	457 (206)		538 (151)	282 (80)	153 (41)	73 (14)	85 (14)
05824	62 (25)	150 (21)	130 (42)	50 (15)	112 (37)	38 (11)	48 (11)
11368	102 (41)	458 (44)	247 (62)	104 (28)	191 (49)	78 (20)	96 (20)
15848	124 (49)	303 (35)	221 (63)	103 (27)	179 (51)	100 (26)	114 (26)
15462	127 (50)	325 (37)	237 (67)	112 (29)	191 (54)	109 (28)	123 (28)
04267	87 (34)	146 (20)	137 (43)	54 (15)	131 (41)	51 (14)	63 (14)
07113	695 (296)		2409 (456)	1527 (302)	311 (73)	229 (48)	241 (48)
03864	272 (121)		353 (101)	177 (50)	143 (41)	63 (14)	75 (14)
15484	181 (71)	657 (62)	394 (96)	189 (47)	277 (68)	174 (42)	194 (42)
15461	146 (58)	628 (54)	651 (109)	283 (51)	241 (59)	140 (33)	160 (33)
11160	309 (138)		390 (111)	198 (56)	145 (41)	65 (14)	77 (14)
06856	306 (138)		390 (112)	198 (57)	139 (41)	59 (14)	71 (14)
06458	827 (376)		908 (251)	492 (140)	173 (41)	93 (14)	105 (14)
13710	420 (189)		501 (141)	261 (74)	151 (41)	71 (14)	83 (14)
06808	525 (240)		609 (172)	321 (93)	145 (41)	65 (14)	77 (14)
04338	470 (206)		1066 (207)	563 (135)	213 (51)	95 (22)	115 (22)
04358	1006 (444)		3580 (559)	2021 (433)	757 (99)	345 (58)	401 (58)

$Query\ Q$ id	$A = sha(Q)$	$det(A)$	$B = det(A \times S)$	$C = det_S(A)$	$B' = mini(B)$	$C' = mini(C)$	$nwa(C')$
13632	132 (58)	339 (33)	248 (66)	113 (33)	128 (36)	47 (9)	55 (9)
01847	559 (252)		1013 (223)	543 (137)	194 (48)	92 (19)	108 (19)
05219	698 (315)		1192 (260)	651 (164)	196 (48)	94 (19)	110 (19)
05226	920 (417)		1558 (338)	867 (218)	208 (48)	106 (19)	122 (19)
03325	753 (342)		834 (231)	450 (128)	169 (41)	89 (14)	101 (14)
03410	938 (427)		1019 (281)	555 (158)	179 (41)	99 (14)	111 (14)
03407	716 (325)		797 (221)	429 (122)	167 (41)	87 (14)	99 (14)
04245	901 (410)		982 (271)	534 (152)	177 (41)	97 (14)	109 (14)
04953	938 (427)		1019 (281)	555 (158)	179 (41)	99 (14)	111 (14)
05463	204 (86)	1180 (70)	332 (77)	152 (38)	180 (48)	78 (20)	96 (20)
12960	167 (68)	1340 (81)	421 (88)	190 (46)	317 (64)	146 (33)	176 (33)
12961	166 (68)	1318 (80)	417 (87)	186 (45)	313 (63)	142 (32)	172 (32)
09123	164 (64)	705 (59)	358 (90)	175 (43)	265 (66)	164 (40)	186 (40)
12514	182 (77)	2734 (112)	320 (77)	146 (38)	247 (57)	114 (28)	140 (28)
12964	128 (52)	560 (48)	277 (67)	120 (31)	219 (53)	96 (24)	118 (24)
08632	128 (52)	629 (51)	277 (67)	120 (31)	219 (53)	96 (24)	118 (24)
12962	129 (52)	576 (49)	281 (68)	124 (32)	223 (54)	100 (25)	122 (25)

The column for $\det(A)$ contains the statistics for the determinization of A . No schema is used there. We use a timeout of 100 seconds. Whenever this is not enough, the cell in the table is left blank. Indeed, the determinization fails with this timeout for 37% of the queries of our corpus. Roughly, the determinization fails for all SHAs with more than 100 states. For instance, for query 11780 the SHA A has has size 205 (88), while the dSHA $\det(A)$ has size 3832 (190).

The column for $B = \det(A \times S)$ contains the determinization of the product of A and the schema $S = xml\&one^x$. Even though $A \times S$ is always larger than A , we were able to always determinize $A \times S$ within the timeout, in contrast to A . The largest dSHA B obtained is for for query 04358: it has size 3580 (559). This show that B may still be quite big, but often a big improvement in size over $\det(A)$.

The next column reports on $C = det_S(A)$ obtained by schema-based determinization with schema $S = xml\&one^x$. Again, the computation succeeds in all cases within the timeout of 100 seconds. The size of C for query 04358 is 2021 (433), which improves in size over B .

In the next two columns, we respectively minimize the determinized SHAs B and C , using a naïve minimization algorithm. All automata can be minimized

within the timeout of 100 seconds. We note that $C' = mini(C)$ is always smaller than $B' = mini(B)$, showing that schema-based determinization yields smaller minimal automata than determinizing the schema-product. The maximal number of states of the minimal dSHAs $C' = mini(C)$ is 58 for query 04358. In average the number of states decreases by 55%.

In the last column, we compiled the minimized dSHAs of C' to the deterministic NWA $nwa(C')$. It has the same number of states than C' for all queries and a minor increase is the number of transitions. All these results, including the automata of the intermediate steps, generated during the whole compilation chain are available at in the software heritage archive at the following url: https:// archive.softwareheritage.org/ browse/ origin/ ?origin_url=https:// gitlab.inria.fr/ aalserha/xpath-benchmark.

8. Conclusion

We provide a benchmark of deterministic automata for regular XPath queries obtained with an algorithm for schema-based determinization of SHAs that we presented. Our benchmark is compiled from forward navigational XPath queries: the 79 largest queries modulo renaming of the 4500 forward navigational XPath queries of the corpus of Lick and Schmitz [30]. From the SHAs of these 79 queries, 37% cannot be determinized in less then 100 seconds by schema-less determinization. Schema-based determinization, in contrast, succeeds for 100% of them. Furthermore, all dSHAs obtained by schema-based determinization are sufficiently small so that they can be minimized with the naïve quadratic algorithm. This leads us to a collection of minimal dSHAs with an average number of states of 22, and 71 as the average number of transition rules.

We hope that the automata of our collection will be used for experimenting with algorithms for XPath queries in the near future and for developing and comparing the performance of algorithms for answering XPath queries on XML streams in particular.

Bibliography

[1] Alur, R.: Marrying words and trees. In: 26th ACM Symposium on Principles of Database Systems. pp. 233--242. 2007.

[2] Bagan, G.: MSO queries on tree decomposable structures are computable with linear delay. In: Comput. Sci. Logic. LNCS, vol. 4646, pp. 208--222. 2006.

[3] Von Braunmühl, B., Verbeek, R.: Input driven languages are recognized in log n space. In: Theory of Computation, North-Holland Mathematics Studies, vol. 102, pp. 1 -- 19, 1985.

[4] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. http://tata.gforge.inria.fr. 2007.

[5] Courcelle, B.: Linear delay enumeration and monadic second-order logic. Discrete Applied Mathematics, 157(12), 2675--2700, 2009.

[6] Debarbieux, D., Gauwin, O., Niehren, J., Sebastian, T., Zergaoui, M.: Early nested word automata for xpath query answering on XML streams. Theor. Comput. Sci. 578, 100--125 (2015).

[7] Muñoz, M., Riveros, C.: Streaming query evaluation with constant delay enumeration over nested documents. International Conference on Database Theory (ICDT). 2022.

[8] Fagin, R., Kimelfeld, B., Reiss, F., Vansummeren, S.: Document spanners: A formal approach to information extraction. J. ACM 62(2), 12:1--12:51 (2015).

[9] Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. 18(2), 194--211 (1979).

[10] Franceschet, M.: XPathmark performance test. https://users.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html, accessed: 2020-10-25.

[11] Gauwin, O., Niehren, J., Tison, S.: Earliest query answering for deterministic nested word automata. International Conference on Fundamental of Computing Theory. LNCS, vol. 5699, pp. 121--132. (2009).

[12] Libkin, L., Martens, W., Vrgoçc, D.: Querying graph databases with xpath. In: International Conference on Database Theory (ICDT) 2013. p.129–140.

[13] Lick, A.: Logique de requêtes à la XPath : systèmes de preuve et pertinence pratique. Theses, Université Paris-Saclay. (2019).

[14] Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and complexity of XML Schema. ACM TODS 31(3), 770--813. (2006).

[15] Martens, W., Trautner, T.: Evaluation and Enumeration Problems for Regular Path Queries. In: International Conference on Database Theory (ICDT) 2018. LIPIcs, vol. 98, pp. 19:1--19:21.

[16] Mozafari B., Zeng, K., and Zaniolo C.. High-performance complex event processing over XML streams. SIGMOD Conference, ACM, 253--264, 2012.

[17] Niehren, J., Sakho, M.: Determinization and Minimization of Automata for Nested Words Revisited. Algorithms. 14(3): 68, 2021.

[18] Niehren, J., Sakho, M., Al Serhali, A.: Schema-Based Automata Determinization. https://hal.inria.fr/hal-03536045. Inria Lille, 2022.

[19] Okhotin, A., Salomaa, K.: Complexity of input-driven pushdown automata. SIGACT News 45(2), 47--67 (2014).

[20] Schmid, M.L., Schweikardt, N.: A Purely Regular Approach to Non-Regular Core Spanners. In: International Conference on Database Theory (ICDT). LIPIcs, vol. 186, pp. 4:1--4:19, 2021.

[21] Seidl, H.: Deciding equivalence of finite tree automata. STACS. LNCS, vol. 349, pp. 480--492. (1989)

[22] Straubing, H.: Finite Automata, Formal Logic, and Circuit Complexity. Progress in Computer Science and Applied Series, Birkhäuser, 1994.

[23] Alur, R. and Madhusudan, P. Adding nesting structure to words. Journal of the ACM, 56(3):1--43, 2009.

[24] Kay, M. The Saxon XSLT and XQuery processor. Available at `https://www.saxonica.com` since 2004.

[25] Labath P. and Niehren J.. A Uniform Programming Language for Implementing XML Standards. In SOFSEM 2015.

[26] Gauwin O. Streaming Tree Automata and XPath. PhD thesis, Université Lille 1, 2009.

[27] Genevès P. and Layaïda N. A System for the Static Analysis of XPath. ACM Trans. Inf. Syst., October 2006.

[28] Gottlob G., Koch C. and Pichler R. The complexity of XPath query evaluation. In 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2003.

[29] Sakho M. Certain Query Answering on Hyperstreams. Phd Thesis. Université de Lille; Inria, 2020

[30] Lick, A., Schmitz S.: XPath Benchmark. Available online at `https://archive.softwareheritage.org/browse/directory/1ea68cf5bb3f9f3f2fe8c7995f1802ebadf17fb5` . Last visited April 13th 2022.

A. Complementary Information

Table A.1. The 79 XPath queries selected from Lick's and Schmitz's corpus

Id	XPath Query
18330	`/ descendant-or-self::node()/child::parts-of-speech`
17914	`/ descendant-or-self::node()/ child::tei:back/ descendant-or-self::node()/child::tei:interpGrp`

Id	XPath Query
10745	*//tei:imprint/tei:date[@type='access']
02091	* | .//reftd
00744	.//@id | .//@xml:id
12060	.//attDef
02762	.//authorgroup/author | .//author
06027	.//authorinitials | .//author
02909	.//bibliomisc[@role='serie']
06415	.//email | address/otheraddr/ulink
03257	.//equation[title or info/title]
05122	.//procedure[title]
09138	.// rng:ref | .// tei:elementRef | .// tei:classRef | .// tei:macroRef | .//tei:dataRef
05460	.//table//footnote | .//informaltable//footnote
12404	.//tei:dataRef[@name]
10337	.//tei:note[@place='end']
06639	.//tgroup//footnote
14340	//*
13804	//GAP/@DISP
13896	//HEADER//IDNO[@TYPE='evans citation']
02194	//annotation
06726	//doc:table | //doc:informaltable
13640	//equiv[@filter]
05735	//glossary[@role='auto']
15766	//h:body/h:section[@data-type='titlepage']
15524	//h:section[@data-type='titlepage']
06512	//reftd//text()
06176	//set | //book | //part | //reference | //preface | //chapter | //appendix | //article | //colophon | //reftd | //section | //sect1 | //sect2 | //sect3 | //sect4 | //sect5 | //indexterm | //glossary | //bibliography | //*[@id]
12539	//tei:elementSpec | //tei:classSpec[@type='atts']
11780	//tei:ref[@type='cite'] | //tei:ptr[@type='cite']

Id	XPath Query
11478	//xhtml:p[@class]
11227	/tei:TEI/tei:text//tei:note[@type='action']
05684	@abbr | @align | @axis | @bgcolor | @border | @cellpadding | @cellspacing | @char | @charoff | @class | @dir | @frame | @headers | @height | @id | @lang | @nowrap | @onclick | @ondblclick | @onkeydown | @onkeypress | @onkeyup | @onmousedown | @onmousemove | @onmouseout | @onmouseover | @onmouseup | @rules | @scope | @style | @summary | @title | @valign | @valign | @width | @xml:id | @xml:lang
06947	anchor | areaset | audiodata | audioobject | beginpage | constraint | indexterm | itermset | keywordset | msg | doc:anchor | doc:areaset | doc:audiodata | doc:audioobject | doc:beginpage | doc:constraint | doc:indexterm | doc:itermset | doc:keywordset | doc:msg
06794	articleinfo | chapterinfo | bookinfo | doc:info | doc:articleinfo | doc:chapterinfo | doc:bookinfo
06169	article | preface | chapter | appendix | reftd | section | sect1 | glossary | bibliography
06924	authorblurb | formalpara | legalnotice | note | caution | warning | important | tip | doc:authorblurb | doc:formalpara | doc:legalnotice | doc:note | doc:caution | doc:warning | doc:important | doc:tip
11958	biblStruct//note
01705	book | article | part | reference | preface | chapter | bibliography | appendix | glossary | section | sect1 | sect2 | sect3 | sect4 | sect5 | reftd | colophon | bibliodiv[title] | setindex | index
02086	book | article | topic | part | reference | preface | chapter | bibliography | appendix | glossary | section | sect1 | sect2 | sect3 | sect4 | sect5 | reftd | colophon | bibliodiv[title] | setindex | index
02000	chapter | appendix | epigraph | warning | preface | index | colophon | glossary | bibliotd | bibliography | dedication | sidebar | footnote | glossterm | glossdef | bridgehead | part
02697	chapter | appendix | preface | reference | reftd | article | topic | index | glossary | bibliography
14183	content//rng:ref

Id	XPath Query
07106	dbk:appendix | dbk:article | dbk:book | dbk:chapter | dbk:part | dbk:preface | dbk:section | dbk:sect1 | dbk:sect2 | dbk:sect3 | dbk:sect4 | dbk:sect5
05824	descendant-or-self::*
11368	descendant-or-self::tei:TEI/tei:text/tei:back
15848	descendant::*[@class='refname']
15462	descendant::h:span[@data-type='footnote']
04267	descendant::label
07113	following-sibling::*[self::dbk:appendix | self::dbk:article | self::dbk:book | self::dbk:chapter | self::dbk:part | self::dbk:preface | self::dbk:section | self::dbk:sect1 | self::dbk:sect2 | self::dbk:sect3 | self::dbk:sect4 | self::dbk:sect5] | following-sibling::dbk:para[@rnd:style = 'bibliography' or @rnd:style = 'bibliography-title' or @rnd:style = 'glossary' or @rnd:style = 'glossary-title' or @rnd:style = 'qandaset' or @rnd:style = 'qandaset-title']
03864	guibutton | guiicon | guilabel | guimenu | guimenuitem | guisubmenu | interface
15484	h:pre[@data-type='programlisting']//text()
15461	h:table[descendant::h:span[@data-type='footnote']]
11160	html:table | html:tr | html:thead | html:tbody | html:td | html:th | html:caption | html:li
06856	imageobject | imageobjectco | audioobject | videoobject | doc:imageobject | doc:imageobjectco | doc:audioobject | doc:videoobject
06458	info | reftdinfo | referenceinfo | refsynopsisdivinfo | refsectioninfo | refsect1info | refsect2info | refsect3info | setinfo | bookinfo | articleinfo | chapterinfo | sectioninfo | sect1info | sect2info | sect3info | sect4info | sect5info | partinfo | prefaceinfo | appendixinfo | docinfo
13710	persName | orgName | addName | nameLink | roleName | forename | surname | genName | country | placeName | geogName
06808	personname | surname | firstname | honorific | lineage | othername | contrib | doc:personname | doc:surname | doc:firstname | doc:honorific | doc:lineage | doc:othername | doc:contrib

Id	XPath Query
04338	refsynopsisdiv/ title | refsection/ title | refsect1/ title | refsect2/title | refsect3/title | refsynopsisdiv/info/title | refsection/ info/ title | refsect1/ info/ title | refsect2/ info/ title | refsect3/info/title
04358	section/title | simplesect/title | sect1/title | sect2/title | sect3/title | sect4/title | sect5/title | section/info/title | simplesect/info/title | sect1/info/title | sect2/info/title | sect3/ info/ title | sect4/ info/ title | sect5/ info/ title | section/ sectioninfo/ title | sect1/ sect1info/ title | sect2/ sect2info/ title | sect3/ sect3info/ title | sect4/ sect4info/ title | sect5/sect5info/title
13632	self::placeName | self::persName | self::district | self::settlement | self::region | self::country | self::bloc
01847	set | book | part | preface | chapter | appendix | article | reference | reftd | book/glossary | article/glossary | part/ glossary | bibliography | colophon
05219	set | book | part | preface | chapter | appendix | article | topic | reference | reftd | book/glossary | article/glossary | part/ glossary | book/ bibliography | article/ bibliography | part/bibliography | colophon
05226	set | book | part | preface | chapter | appendix | article | topic | reference | reftd | sect1 | sect2 | sect3 | sect4 | sect5 | section | book/ glossary | article/ glossary | part/ glossary | book/ bibliography | article/ bibliography | part/ bibliography | colophon
03325	set | book | part | reference | preface | chapter | appendix | article | topic | glossary | bibliography | index | setindex | reftd | sect1 | sect2 | sect3 | sect4 | sect5 | section
03410	set | book | part | reference | preface | chapter | appendix | article | topic | glossary | bibliography | index | setindex | reftd | refsynopsisdiv | refsect1 | refsect2 | refsect3 | refsection | sect1 | sect2 | sect3 | sect4 | sect5 | section
03407	set | book | part | reference | preface | chapter | appendix | article | glossary | bibliography | index | setindex | reftd | sect1 | sect2 | sect3 | sect4 | sect5 | section

Id	XPath Query
04245	set | book | part | reference | preface | chapter | appendix | article | glossary | bibliography | index | setindex | reftd | refsynopsisdiv | refsect1 | refsect2 | refsect3 | refsection | sect1 | sect2 | sect3 | sect4 | sect5 | section
04953	set | book | part | reference | preface | chapter | appendix | article | glossary | bibliography | index | setindex | topic | reftd | refsynopsisdiv | refsect1 | refsect2 | refsect3 | refsection | sect1 | sect2 | sect3 | sect4 | sect5 | section
07095	sf:stylesheet | sf:stylesheet-ref | sf:container-hint | sf:page-start | sf:br | sf:selection-start | sf:selection-end | sf:insertion-point | sf:ghost-text | sf:attachments
05463	table//footnote | informaltable//footnote
12960	tei:classSpec/tei:attList//tei:attDef/tei:datatype/rng:ref
12961	tei:classSpec/tei:attList//tei:attDef/tei:datatype/tei:dataRef
09123	tei:content//rng:ref[@name = 'macro.anyXML']
12514	tei:content/ tei:classRef | tei:content// tei:sequence/ tei:classRef
12964	tei:dataSpec/tei:content//tei:dataRef
08632	tei:front//tei:titlePart/tei:title
10595	tei:label | tei:figure | tei:table | tei:item | tei:p | tei:title | tei:bibl | tei:anchor | tei:cell | tei:lg | tei:list | tei:sp
12962	tei:macroSpec/tei:content//rng:ref

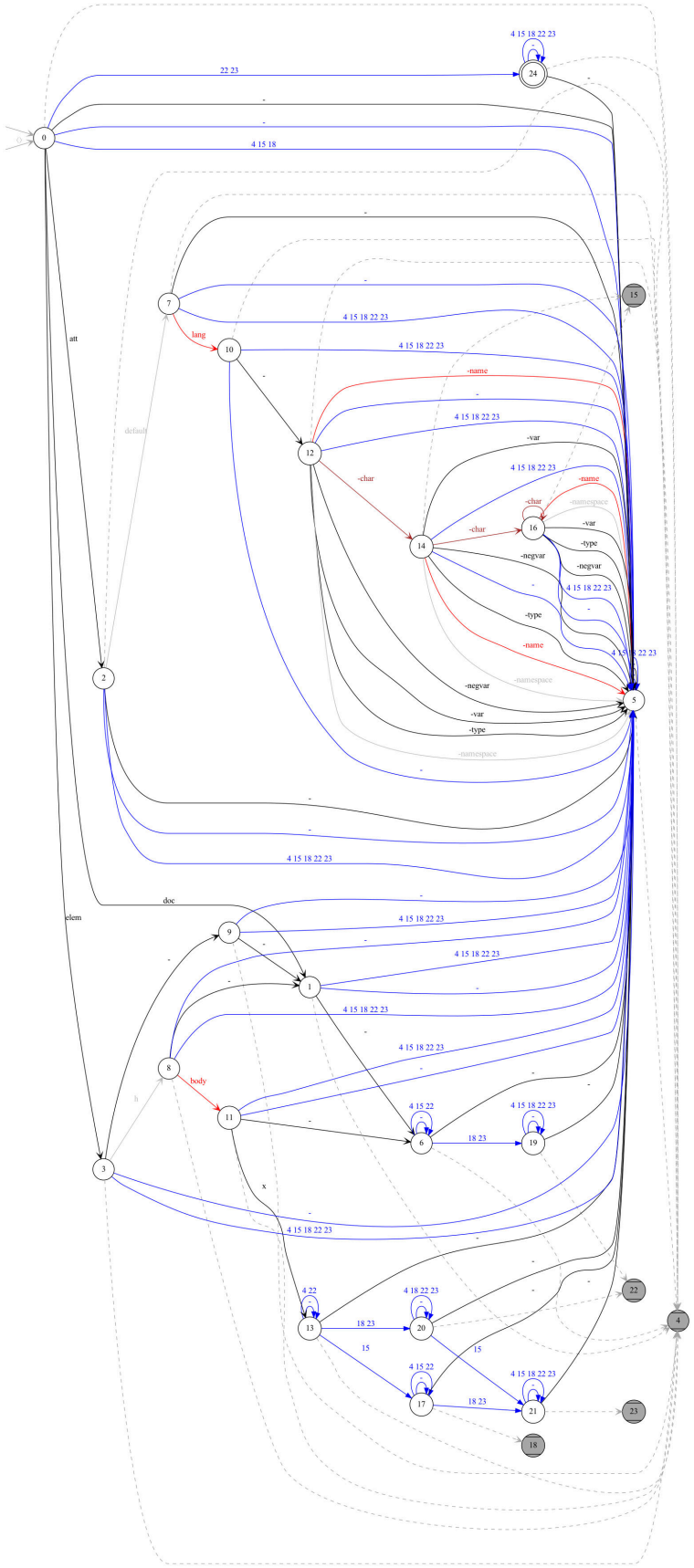


Figure A.1. The determinization $\det(A_2)$ of the SHA A_2

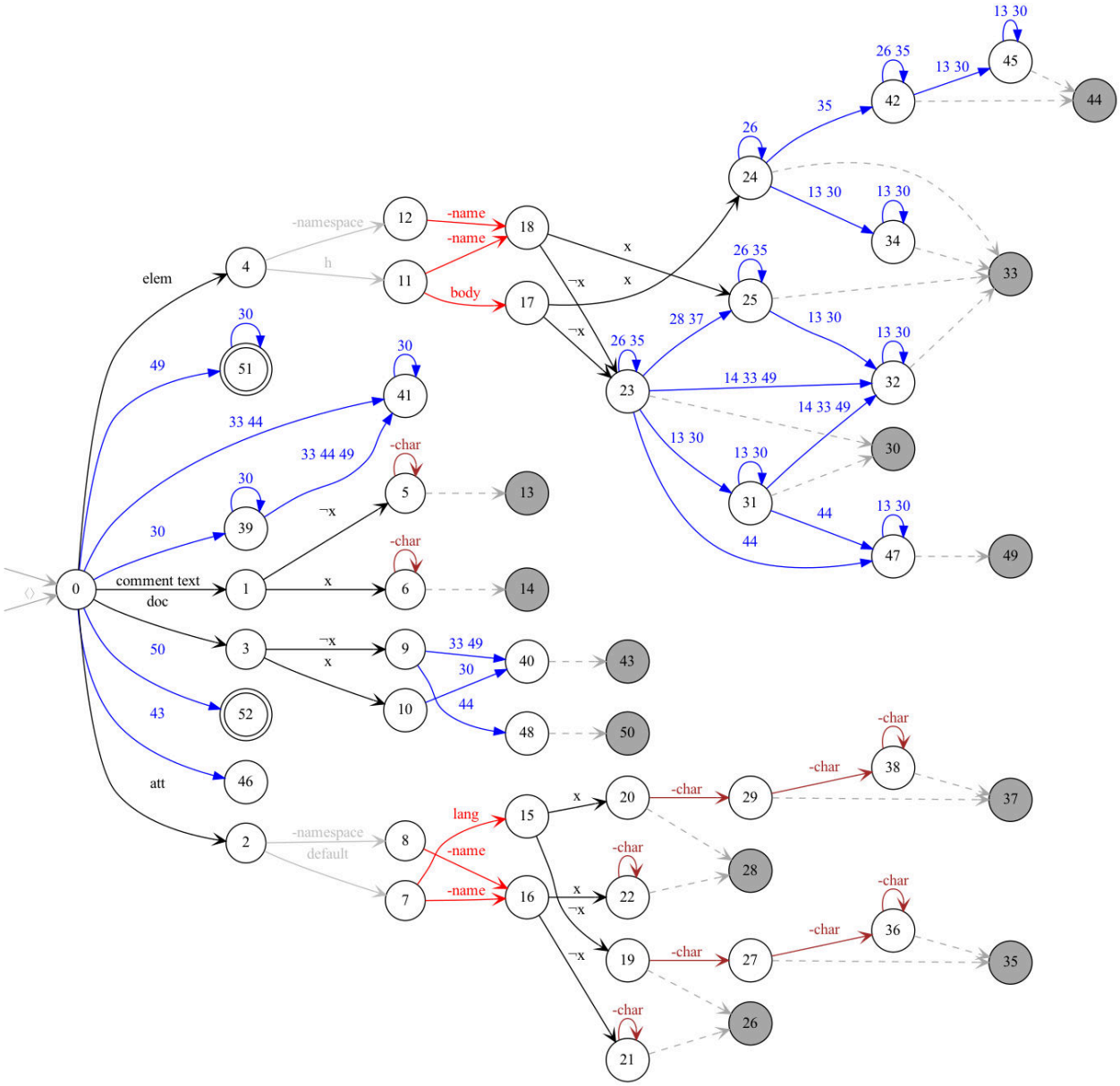


Figure A.2. The determinization of the schema product $\det(A_2 \times xml\&one^x)$

Use the Markup, Stupid!

Ari Nordström

Abstract

The XML technology stack has been around for more than 20 years and is, by all accounts, mature. Why is it that some insist on non-XML solutions for XML problems? Use the markup, stupid!

1. The Why

This paper is sort of a rant and has been a long time coming. Allow me to explain.

Those of us who've been in the markup business for some time will appreciate the relative order and maturity of the XML technology stack. We have at our disposal well-established and mature tools during the entire process, from authoring to storing to publishing. We have XML standards that help us write content, from DocBook to DITA to S1000D and beyond, all of which we can reasonably expect to be supported in our chosen editor. We have XML databases that not only store our content natively, as XML, but that allow us to easily query any content stored. And we have XML transformation standards and processors that help us realise, what long was a pipe dream, single-source publishing to any number of output formats, be it PDF, responsive design HTML on a mobile phone, or Rich Text Format.

Why, then, would anyone willingly bring in outdated non-markup technologies to manage XML when the XML-based solution is better in every way?

Enter a well-known and widely spread product in the Product Life Management (PLM) space. Like other PLM products, this one is intended to manage all aspects of product information. The engineering data, known as "CAD data", lives there and can then be coupled with product documentation, from parts catalogues to maintenance tasks, and so on.

Standards like DITA and S1000D are common in this space, as they are intended to be written in topic-size chunks, each topic focussing on a specific and narrow subject such as an assembly or disassembly, or perhaps a reference topic describing a component. Both define different topic types to cover different content; DITA, for example, includes topic types such as "concept", intended to describe a product, and "task", for step-by-step instructions.

The PLM product supports coupling it with different XML editors, including a highly specialised editor that allows us to create procedures based directly on the CAD data and then use that to *generate* matching DITA or S1000D, including illustrations in 2D or 3D, also generated from the CAD data.

Both standards also use a separate document type to describe how the topics are organised to form a complete deliverable manual. DITA, for example, uses "maps", XML documents that link to, and organise, topics in document hierarchies.

This is where the problems start in earnest.

2. A Quick DITA Refresher

It could well be that you don't know DITA or can't remember much of it. I fully understand and sympathise. This section is a quick refresher, giving you only the bare bones, the basics, of DITA, and mostly those things called maps.

So, DITA topics come in various types, but essentially they all have a title, some metadata, and a body with the actual content:

```
<topic id="topic_fjt_3bt_2tb">
    <title>Topic A</title>
    <prolog>
        <author>Mark Up</author>
    </prolog>
    <body>
        <p>Topic content.</p>
    </body>
</topic>
```

They are intended to describe one topic, whatever it may be[1], and they are meant to be reusable. The reuse is defined in a *map*, and while there's more to them than that, again the basic are straight-forward. This map, for example, organises topics in a chapter-and-section hierarchy:

```
<map id="id-map-AtoE-nested">
    <title>Topicrefs A to E Nested</title>
    <topicref href="topicA.dita" navtitle="tref A, level 1">
        <topicref href="topicB.dita" navtitle="tref B, level 2">
            <topicref href="topicD.dita" navtitle="tref D, level 3"/>
            <topicref href="topicE.dita" navtitle="tref E, level 3"/>
        </topicref>
        <topicref href="topicC.dita" navtitle="tref C, level 2"/>
    </topicref>
</map>
```

Published with a DITA-compliant stylesheet, the resulting table of contents might look like this:

[1] And the trick tends to be to define the right size and scope for the topics.

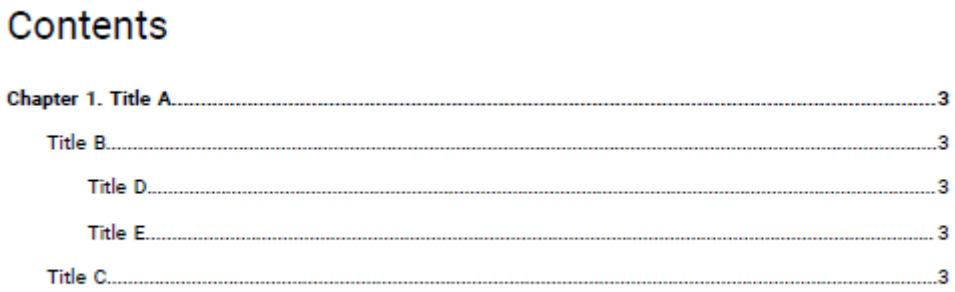


Figure 1. Nested Topicrefs Table of Contents

A different DITA map might reuse the same topics in a different way. This, for example, publishes topics A and B as sibling sections:

```
<map id="id-map-AB-siblings">
    <title>Topicrefs A and B As Siblings</title>
    <topicref href="topicA.dita" navtitle="tref A, level 1"/>
    <topicref href="topicB.dita" navtitle="tref B, level 1"/>
</map>
```

In other words, DITA topics contain the building blocks with the content while DITA maps organise that content in documents. Obviously there is more to the standard, but this is quite enough for our purposes.

3. The PLM Product

The PLM product, built aroud a SQL database, includes a content management module that can be connected to an external editor, some better integrated than others. At a minimum, you can configure a text editor to open your checked-out documents. There is a publishing module that can produce any output, from PDF with XSL-FO engines such as Apache FOP or Antenna House Formatter, to HTML, as long as you can define an Ant build script for your publishing process.

The product includes version handling, impact analysis (the ability to trace the use of a database item, be it a document or something else, and various other features. To put it succintly, the product is *huge* and endlessly expandable. So what's there not to like?

The content management module is geared towards topic-based publishing, and here, we'll focus specifically on DITA even though the discussion here is equally valid for S1000D or some other standard. Topics are, of course, authored using the external editor and then checked in, versioned, and stored in the database. Maps, however, are a different matter.

You can create and edit the map directly in the content management module interface, meaning that you can copy and paste, and drag and drop, you topics to

the map, reorganising them with a few clicks. As you'd expect there is a nice tree-based view that you'll use, not only to edit the map but to check out and edit the topics linked to it:

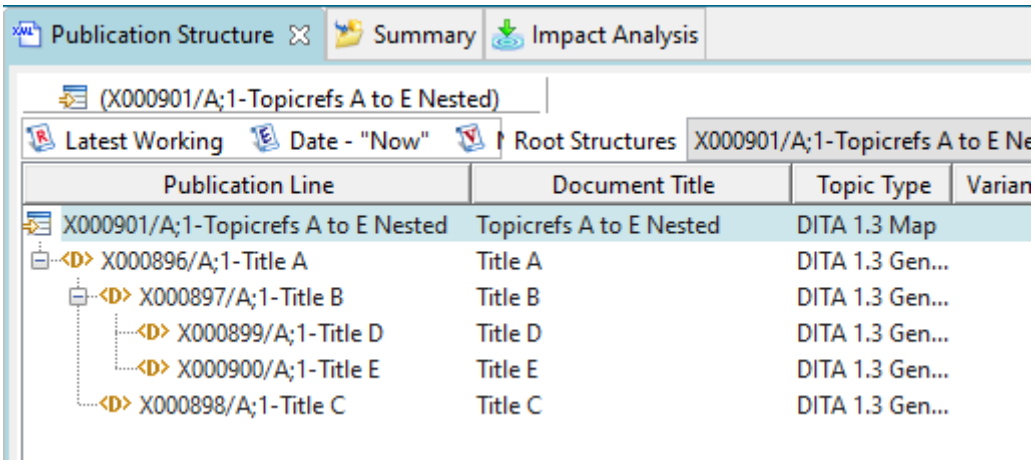


Figure 2. PLM Publication Structure View

You might recognise this as the previous section's DITA map.

Publishing and editing are both available via right-clicking, as are a number of other functions. While the client GUI may not be the most up-to-date here, there is a web-based version that offers a much nicer and modern interface. *What's there not to like?*

4. Importing (and Exporting) Maps

When you import DITA to the system, you start with the leaf nodes. Images are first, followed by topics and, finally, maps. The import is unable to pull in a map and follow the links to import linked resources until done, and the explanation is a rather surprising one:

Maps, internally in the system, are not XML at all. They are database objects that are related to other database objects, and there is a mapping process that defines how an import of a map should break down or *decompose* the map. Thus, that nice tree view where you can drag, drop, copy and paste topics, is a representation not of the XML but of the internal relational model.

The decompose process is governed by rules that are defined in an administrative user interface where the various database objects are mapped to or from XML nodes using functions manipulating either the object or the XML node. The XML node is addressed using XPath-like expressions[2] that declare what you are reading from or writing to—and here is the kicker:

[2]If you are an XML person, this will get you at first because you try to think in terms of XPaths and you really shouldn't, because you will be disappointed.

When importing content, be it the first time or the 12th, you are *decomposing* that content, based on the mapping rules. The XML is broken down and stored in the database as items with various relations, presentable in the structure view. Similarly, when the content is exported, when publishing and when checking out content for editing, you are *composing* the items, the fragments from the decompose process, from the database into some XML format. You might be round-tripping your content, with only some slight changes and additions as would be the case with DITA topics[3], or you might be building a map from scratch, or rather that non-XML representation stored in the database.

5. So What's Wrong with It?

You might reasonably ask why I'm bringing this up. OK, so here is the thing: DITA uses maps to relate topics to each other in the context of the map. One map might do this:

```
<topicref href="A.dita">
    <topicref href="B.dita"/>
</topicref>
```

This is a two-level hierarchy with topic A starting a section and topic B included as a subsection. In the context of thjis particular map, B is a subsection in A. Another map might look similar to it but include a topic C:

```
<topicref href="A.dita">
    <topicref href="B.dita"/>
    <topicref href="C.dita"/>
</topicref>
```

Here, both B and C are subsections to A in the context of this map. This is not true outside the map, just as B being a subsection to A isn't true outside the context of the first map. Maps define relationships between topics without the topics being aware of the fact. It's a bit like extended XLink, really.

5.1. Maps and Inheritance

In object-oriented programming, and in quite a few (other) abstraction models, the concept of inheritance is a fact of life; properties are passed on from an ancestor to a child, from a generalisation to an instance, etc. It is quite common that programmers who dabble in XML from that perspective[4] expect inheritance in XML trees, and while XML people dabbling in programming sometimes add to the confusion (namespaces, anyone?), the fact of the matter is that XML doesn't work like that. Also, many standards do include inheritance concepts; DITA, for

[3] And I'll get into a bit more detail on this later.
[4] In Ken Holman's terminology, they're known as “codeheads”.

example, relies on it for a variety of constructs. Topic hierarchies, however, do not.

Let's define a map, shall we? This defines a top section (“Chapter One Title”) with some subsections and sub-subsections:

```
<map id="id-singlemap1">
    <title>A to E as Single Map Title</title>
    <topichead navtitle="Chapter One Title">
        <topicref href="topicA.dita"/>
        <topicref href="topicB.dita"/>
        <topichead navtitle="Section One Navtitle">
            <topicref href="topicC.dita"/>
        </topichead>
        <topichead navtitle="Section Two Navtitle">
            <topicref href="topicD.dita"/>
            <topicref href="topicE.dita"/>
        </topichead>
    </topichead>
</map>
```

Note the last subsection, “Section Two Navtitle”, with its two topic references to D and E. We import this to the PLM system and get this publication structure:

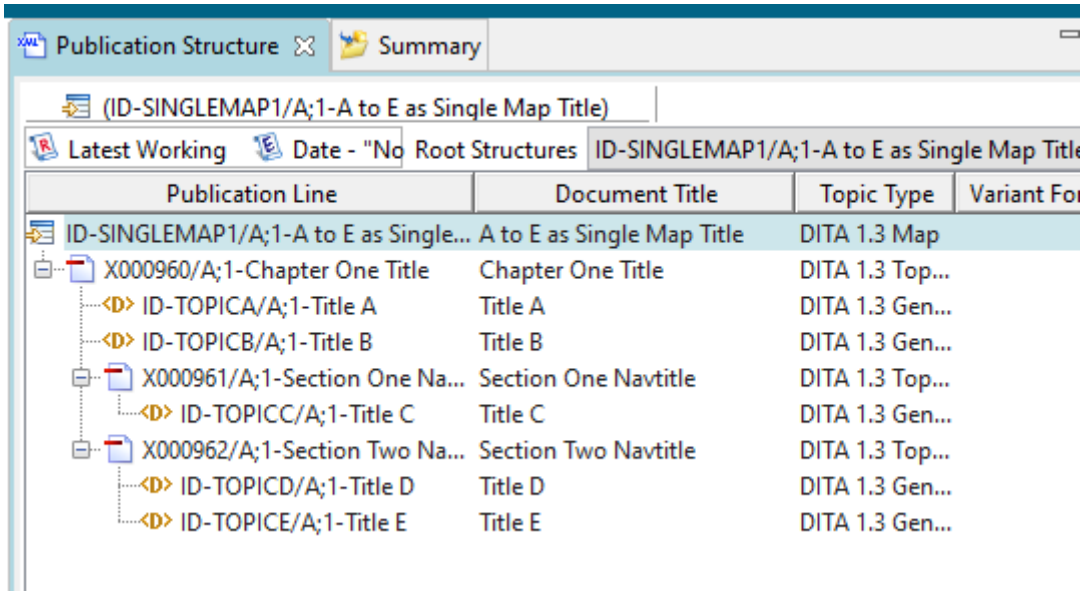


Figure 3. Map 1

Let's now define a second map based on the previous one:

```
<map id="id-singlemap2">
    <title>A to E as Single Map Title</title>
    <topichead navtitle="Chapter One Title">
        <topicref href="topicA.dita"/>
```

```
        <topicref href="topicB.dita"/>
        <topichead navtitle="Section One Navtitle">
            <topicref href="topicC.dita"/>
        </topichead>
        <topichead navtitle="Section Two Navtitle">
            <topicref href="topicD.dita"/>
            <topicref href="topicE.dita">
                <topicref href="topicF.dita"/>
            </topicref>
        </topichead>
    </topichead>
</map>
```

They're almost the same; the second map has an added nested `topicref` to topic F, last. We import this map and get this tree:

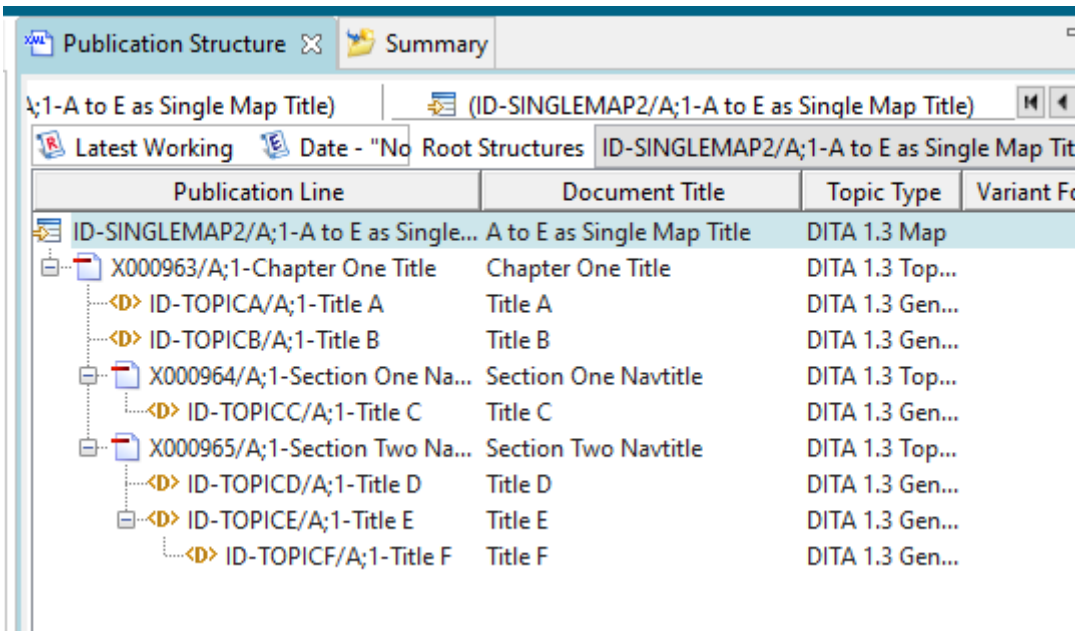


Figure 4. Map 2, with an Additional Nested Topicref

The added relation is easily spotted and what we expect. But when we open map 1 again, look what's happened:

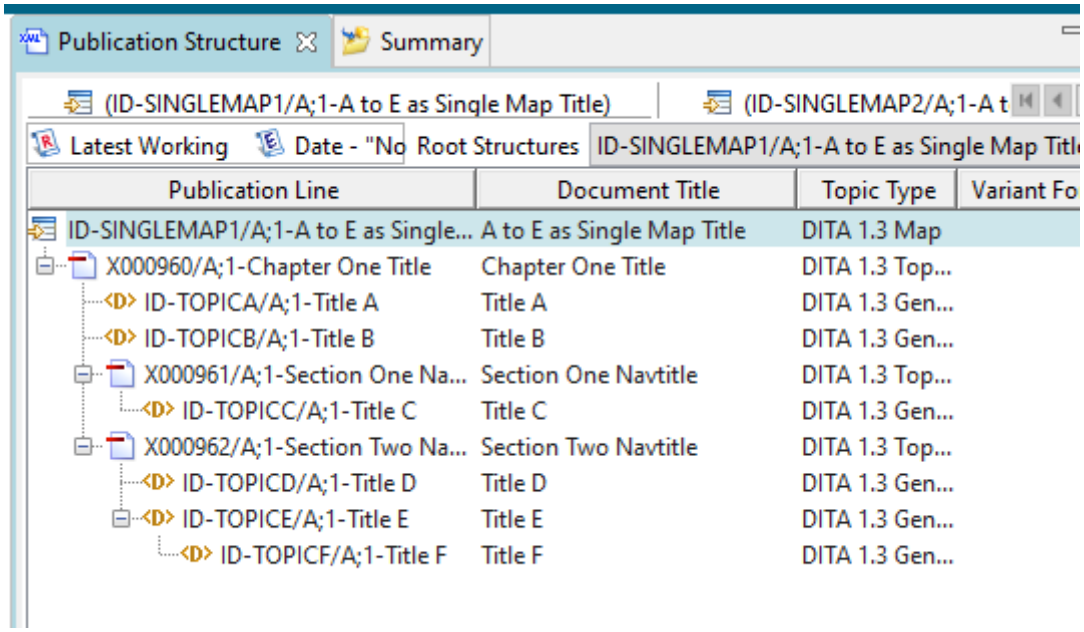


Figure 5. Map 1 Changed After Map 2 Import

The updated nested `topicref` *is interpreted as an updated relation between the topics.* This means that the database representation does not differentiate between the `topicref` and the `topic` it links to.

The problem is a fundamental one, and, as far as I can tell, there is no way around it.

5.2. Decomposing and Composing

The decompose process is defined in the mapping rules. The easiest rules are either "compose" or "decompose", where the function applied on the mapping stipulates that the same process is run in a single direction. This, for example, maps an `id` attribute to the `item_id` database property:

Attribute Name	Function	Path
item_id	Bidirectional	/@id

Figure 6. Mapping @id to item_id Property

But we also have other rules, such as these two mapping a `title` element to the `DocumentTitle` property, on decompose and on first compose[5]:

[5]The first compose rule is used when the map is created in the product rather than in an XML editor.

DocumentTitle	First Compose	/title
DocumentTitle	Decompose	/title

Figure 7. Mapping Document Titles

Why in both directions? Simply because it is possible to edit a title directly in the GUI, outside the XML. If you edit the title, you want it to carry through to an export and vice versa.

A very common mapping rule is to carry over database properties to the XML document by mapping them to processing instructions:

item_id	Bidirectional	/?id

Figure 8. Mapping @id to a Processing Instruction

It is possible to map properties to element and attribute content, but the addressing language is very limited. As an XML person you might be tempted to say something like this:

```
/data[@name="document_no"]/@value
```

Unfortunately, the language does not support predicates or indeed any kind of condition directly. It only does simple addressing, which is why most properties are mapped to PIs directly inside the XML root element and then transferred to the actual XML nodes—elements or attributes, with or without predicates—using XSLT when publishing.

6. Discussion

The unwanted topic relationships—the inheritance problem, as I call it—are the result of a poor underlying model. Poor because it shows the ignorance of the model's designer regarding how DITA is supposed to work, and poor because imposes lmitations on DITA maps, limitations that the source XML format doesn't have.

Personally I suspect that to at least some degree, the flawed design happened because the objective was not to correctly represent a specific XML flavour but to define a model that would be usable for any schema. In DocBook, for example, you'd normally define your section hierarchies inline, in the document, like this:

```
<section>
    <title>Top Level</title>
    <para>Para content.</para>
    <section>
        <title>Section Level</title>
        <para>More content.</para>
```

```
    </section>
</section>
```

Here, the hierarchy is defined as part of the content itself. Decompose and compose rules breaking apart and assembling the document can be triggered on `section` elements. The "Section Level" subsection here would be reusable in other documents after decompose, but if something was added to it, for example, a third-level section, every reused instance of the subsectionwold also get it, which (arguably) makes more sense.

In a DITA implementation, however, the inheritance problem is an unqualified disaster.

The mapping rules are just as flawed. As their capabilities are limited, roundtrips (decompose followed by compose) are very difficult or even impossible, not to mention that the XPath-like syntax will likely only serve to confuse, especially if you know the real XPath.

What we have, then, is a flawed, non-XML representation of an XML document that cannot possibly support the XML's feature set, and to make matters worse, the language and methods used to get us from the XML to that representation and back is not sufficient.

7. How Did This Happen?

I guess the remaining question is "what went wrong?" or maybe just "why, oh why would they do something like this?" I can certainly see how:

- That publication structure view is built on top of a tree representation meant for viewing non-XML content—engineering data, most likely—in the PLM system, and someone thought that it resembles a document hierarchy, so wouldn't it be nice if...?
- DITA maps look like that. S1000D publication modules look like that. And DocBook articles look like that. Really, isn't all XML hierarchical data...?
- That tree view can be used to represent *any* XML. All we need is a way to translate the XML to it and back.
- Show a tree to a non-XML developer and inheritance will follow.

Of course, I am guessing, but it does sound plausible, doesn't it? There's also that relational models are terrific for keeping track of engineering data—spare parts, kits, assemblies—but often less than perfect for representing documents[6].

The XML (and SGML) world used to be full of similar approaches. Consider Adobe FrameMaker, a product that started out as a very capable word processor for technical documentation but that was extended to support SGML and, later,

[6]These days, I'd pick an XML database every time.

XML. The source file format was never structured, though, it was mapped to structures using rules not unlike the mapping rules we've discussed above.

FrameMaker's `.fm` files were no more related to SGML than the PLM product's internal tree representation is related to XML.

In fact, jumping on the SGML (and XML) bandwagon was all the rage for a time. There was a version of WordPerfect that supported SGML. I remember at least two plugins for Microsoft Word attempting something similar, and, of course, Word would later introduce its own XML format. Et cetera.

It's a greyscale, though; XML's early history is partly the history of tools, languages, and standards trying to incorporate XML because it was the cool new thing. Eventually, as the technology matured, many realised XML everywhere was a bad idea while others continued to use it where it actually works. Like in documents.

8. Conclusions

Which brings us back to where we started. The XML technology stack is quite mature, and the tools are all there. They easily support the document standards mentioned in this little treatise, and more. I am writing this in oXygen XML Editor, in DocBook, but oXygen also offers full DITA support, alongside many other documentation standards, not to mention that it's easy to add further document types.

I've mentioned XML databases. There are several out there, from open-source implementations to enterprise-level commercial products, and we can easily make them support all the XML document types we need.

I don't need to mention all those other technologies available to us again, do I?

I should mention that I don't disapprove of the PLM product as such; the ability to manage your engineering data is extremely useful, especially when coupled with your documentation. I disapprove of the way XML is integrated with the product.

So, a few comments and complaints:

- The tree representation wasn't XML, and what's worse, its internal model wasn't even close. Representing XML in a non-XML way is a bad idea because it will only ever be as good as the mapping rules from and to XML.
- The designers didn't understand XML, only their domain. Inheritance may be a core feature of whatever the tree representation is based on, but the failure to recognise is the designer's, not the underlying feature.
- The fact that the tree representation is not XML requires that XML is mapped to and from that representation. This led to the mapping language, likely another misunderstanding by the designers because it clearly got its influen-

ces from XPath but did not even try to include conditionals or any other useful XPath features beyond some passing resemblance to XPath syntax.

- The mapping language, combined with the limits of the tree, risks giving markup languages and DITA in particular a bad reputation and, worse, hindering implementation and forcing people to look outside the XML technology stack.

  DITA, regardless of what you feel about it[7], is a standard intended to ease authoring topic-based information, but also to get away from some aspects of other XML formats considered difficult. Doing a non-XML version of it doesn't make it easier.

Essentially, if you are trying to copy XML features and functionality, why not use XML. To put it simply:

Use the markup, stupid!

[7]The author of this paper isn't too thrilled about it, but that's another story.

XSL-FO/CSS Comparison

Tony Graham
Antenna House, Inc.
<tony@antennahouse.com>

Abstract

Comparing XSL-FO and CSS formatting is not straightforward. XSL implementations are not standing still: XSL formatters are still incrementally improving even though the XSL Recommendation has not been updated since 2006. CSS is definitely not standing still, although some of the modules most relevant to paged media are advancing slowly, if at all, and some paged media features have been removed in more recent Working Drafts.

This is a high-level view of the differences and similarities between XSL-FO and CSS, based on an extensive new analysis by Antenna House that itself is formatted identically using both XSL-FO and CSS. It also covers some of the features of how the two versions are produced.

1. Introduction

Comparing XSL-FO and CSS formatting is not straightforward. XSL implementations are not standing still: XSL formatters are still incrementally improving even though the XSL Recommendation has not been updated since 2006. CSS is definitely not standing still, although some of the modules most relevant to paged media are advancing slowly, if at all, and some paged media features have been removed in more recent Working Drafts.

The first part of this paper compares the specifications for XSL-FO and CSS. It is based on a much longer document that both compares XSL-FO and CSS and provides information about applicable AH Formatter extensions for one or both of XSL-FO and CSS. That document is available in two versions—formatted using XSL-FO and formatted using CSS—for you to compare.

The second part covers some of the features of how the two versions of the analysis document are produced.

2. History

The following table shows some of the significant events in the development of XSL-FO, CSS, and HTML.

Year	XSL-FO	CSS	HTML
1996	DSSSL	CSS 1	
1998		CSS 2	XHTML 1.0
1999		First CSS 3 drafts	
2001	XSL 1.0		XHTML 1.1
2004			WHAT WG formed
2006	• XSL 1.1 • XSL-FO 2.0 Workshop		HTML WG rechartered
2008			
2009	XSL-FO 2.0 Design Notes		
2011		CSS 2.1	
2012			
2018		CSS Snapshot 2018	
2019			W3C cedes HTML5 to WHAT WG
2020		CSS Snapshot 2020	
2021			

Cascading Style Sheets, level 1, 3 became a Recommendation in 1996. CSS was co-invented by Håkon Wium Lie and Bert Bos. CSS 1 built upon previous style sheet proposals, including earlier separate proposals by Lie and Bos. References to the earlier proposals are at: the W3C *Historical Style Sheet proposals* 7 page; the *The CSS saga* 11 chapter from Lie and Bos's CSS book: and Lie's Ph.D thesis 10. The goals for CSS stated in 1995 1 include: "CSS supports stream-based (or 'incremental' formatting) where possible"; "CSS offers both readers and authors control over the style"; as well as "avoiding an uncontrolled growth of HTML extensions".

DSSSL 5, the stylesheet language for SGML, became an International Standard in 1996. DSSSL defines a tree transformation process followed by a formatting process. In practice, the tree transformation process was not widely used. However, the most widely-used DSSSL formatter had an extension for performing a transformation as an alternative to formatting.

Styling for XML was always part of the development of XML. It was referred to by Jon Bosak, original XML Working Group (WG) Chair, in 1997 as "xml-style (Part 3 of the XML specification suite)" 17 and "Part 3 of the W3C XML suite of specifications for the use of SGML, HyTime, and DSSSL subsets on the World Wide Web" 21.

'xml-style' became 'Extensible Stylesheet Language' (XSL). The XSL WG was formed in 1998, and its charter stated its intention "to define a style specification language that covers *at least* the formatting functionality of both CSS and DSSSL." 19 XSL encompasses both transformation and formatting, but transformation

proved generally useful, and the transformation component was broken out as the XSLT series of Recommendations. The bulk of the XSL 1.0 and XSL 1.1 Recommendations concern the formatting objects (FO) and their properties. The transformation component is covered by a short chapter that refers to the then-current XSLT 1.0 Recommendation. The XSL properties align as much as possible with the corresponding CSS properties, in keeping with the commitment in the XSL WG charter.

The need for consistency in properties was stated to be an architectural principle for the web in the *Consistency of Formatting Property Names, Values, and Semantics* TAG Finding 12 published in 2002.

Antenna House first proposed greater compatibility between XSL-FO and CSS, especially compatibility with the CSS 3 drafts, at *International Workshop on the future of the Extensible Stylesheet Language (XSL-FO) Version 2.0* 18 at Heidelberg, Germany, in 2006. That proposal was not supported by the workshop participants.

AH Formatter V5.0, released in 2008, was the first AH Formatter version to support both XSL-FO and CSS. AH Formatter is still the world's only XSL-FO and CSS formatter, and successive releases have added features for both XSL-FO and CSS.

3. Viewpoints

How a user compares XSL-FO and CSS can depend on their initial exposure to markup and styling as much as or more than on the relative merits of either technology. This is an informal summary of how users of CSS can see XSL-FO, and vice-versa.

3.1. CSSer's view of XSL-FO

- *Source XML or HTML must be transformed into the XSL-FO vocabulary*

  Transformation has advantages and disadvantages. CSS was designed to support stream-based or incremental formatting 1, which is part of why CSS selectors cannot match 'down' into the content of the current element or 'forward' to the structure of following elements. A transformation stage, on the other hand, typically (although less so after XSLT 3.0 added streaming for XSLT) requires the whole document to be available, but it does allow style decisions to be made based on the whole document. Transformation also allows the content to be duplicated and reordered to, for example: generate tables of contents and indexes; sort the rows of a table; or calculate subtotals.

  Formatting paged media using CSS 3 typically requires some form of transformation anyway. This includes generating the running elements that are taken out of the flow and used in headers and footers, as well as generating tables of contents and indexes.

- *Separate attributes for each property is verbose*

  XSL-FO is designed to be the result of an XSLT transformation. XSL-FO was not meant to be read, let alone authored, by humans. It does happen, of course. When XSL-FO is serialized as XML, it is straightforwardly usable in XML editors, etc., and it is as human-legible and as reasonably clear as any other XML or HTML document.

  When XSL-FO is generated using XSLT, the XSLT is less verbose than the XSL-FO that it creates. An FO and its properties can be generated from literal elements and attributes inside an XSLT template, but a single XSLT template can be used many times to generate multiple copies of the FO. Attribute sets in the XSLT are defined once and are used in multiple places in the XSLT. An XSLT template can contain more than just literal elements and attributes to copy to the result: the XSLT can include as much logic as is necessary to be able to conditionally generate the correct FOs, their correct properties, and the correct property values for any given context.

- *JavaScript could be used instead of XSLT*

  Yes, JavaScript can be used to generate tables of contents, indexes, and so on by manipulating the DOM of the document, but whatever JavaScript solutions exist have been bespoke code. Over the last 20 years, there hasn't been a standard, widely used, general purpose JavaScript library that matches both the path-matching ability of XPath and the declarative templating mechanism of XSLT. AH Formatter will format a correct XSL-FO file no matter how it was created.

- *XSL-FO properties inherit but do not cascade*

  The role of the cascade is taken by the XSLT transformation.
  In XSLT:

  - One XSLT stylesheet can import another, similarly to how `@import` imports another CSS style sheet
  - The 'match' patterns in XSLT templates have a default priority based on the specificity of their XPath pattern, similarly to more specific CSS selectors overriding more general ones. In addition, an XSLT template may be given an explicit numeric priority.
  - When there is more than one matching template with the same precedence, the one that occurs last is used, similarly to two CSS rules that have the same weight
  - An XSLT 'attribute set' is a named set of attribute definitions. The attribute definitions are reevaluated in each context where the attribute set is used. Multiple <xsl:attribute-set> with the same name are aggregated, with definitions for individual attributes in an <xsl:attribute-set> with higher precedence overriding definitions in other <xsl:attribute-set> that have lower

precedence. An <xsl:attribute-set> can also reuse attribute definitions in other, named attribute sets. Attribute sets are not quite the same as on-the-fly building up of the properties to apply in a particular context based on the cascade of `@import` rules and specificity of CSS selectors, but they do make it easy to apply a group of properties in particular contexts.

- *FOs are like elements with fixed `display` property values*

  In CSS, all that you have is the source document (unless, as noted previously, you have augmented the original source document to create tables of contents, etc.). The `display` property, like all properties, is applied as that document is formatted. With transformation before formatting, the decisions about how to format each part of the source document are part of the transformation, so there is no need for on-the-fly changes using a `display` property.

- *XSL-FO does not have variables*

  CSS gained variables only recently, but variables are not needed in XSL-FO. XSLT has variables (which have scope and which can be passed between templates), so any calculations based on variables or any substitution of constant values can be handled in XSLT.

- *XSL-FO can't be used for both web and print*

  XSL 1.1 defines how to handle a page that extends indefinitely in one or both dimensions and it also includes some interactive FOs, but neither of those has been widely implemented because few users have ever expressed interest in them.

- *CSS is easier to learn that XSL-FO*

  The basics of CSS syntax are easy to learn, but there is an expanding list of CSS selectors and pseudo-elements to be remembered, plus many CSS properties have their own micro-syntax for expressing their value. The 2014 charter for the CSS Working Group states "CSS is a rather large and complex language." 23, while later charters change the narrative to "The CSS specification is large" even as CSS has further expanded. 24

  XSLT and XSL-FO are XML vocabularies, so most of their syntax is easily understood by anyone who can read XML or HTML. However, XSLT and XSL-FO attempt more than CSS, so it is not surprising that there is more to learn. XSLT is a Turing-complete declarative language for specifying transformations, whereas CSS is a declarative language for specifying styles. People who are used to imperative programming languages where they specify every step of the program can have trouble adapting to the XSLT model where the structure of the source document can determine the program flow.

  XSL-FO provides more control than CSS over, for example, the selection of page masters and the content of headers and footers, so it has more FOs and properties for those areas.

- *There are more CSS users than XSL-FO users*

True. However, comparatively few CSS users are familiar with using CSS to generate paged media. If you want to learn more about CSS for paged media, see *Introduction to CSS for Paged Media* 8, available from the Antenna House website.

3.2. XSL-FOer's view of CSS

- *CSS 'just decorates the tree'*

  Decorating the tree of elements fits with the original goal of CSS to support streaming or incremental formatting. 1 XSLT, in contrast, can use any part of the document, or of a different document, when deciding which template to use in the current context. By using modes, XSLT can process the document or parts of the document multiple times in different ways.

  The structure of the XSL-FO document does not need to match the structure of the source document: the XSLT stage can generate literal elements as well as copy all or part of any node in the source document. An XSLT template can select which nodes to process next rather that just processing the children of the current node.

- *CSS selectors won't look 'down' or 'forward'*

  CSS selectors can match on the current element, its class (or classes), its attribute values, its ancestor elements, and its preceding elements. Selectors won't, however, match on the string value of an element or on any aspect of the type and arrangement of an element's descendent elements. In contrast, the entire document (and possibly other, external documents) is available to the XSLT processor, and style decisions can be made based on more than just the context of the current element.

- *CSS only operates on elements*

  CSS selectors only apply to elements. In contrast, XSLT templates can match on text nodes, text nodes with particular values, or text nodes in particular contexts (possibly with particular values) and can generate FOs based on those text nodes. XSLT can similarly match on comments and processing instructions and generate FOs.

4. Feature comparison

The following table provides an overview of the differences between XSL-FO and CSS. For ease of comparison, the sequence in both the table and the rest of this document follows the chapter sequence in *Introduction to CSS for Paged Media* 8.

Table 1. Feature comparison

Section	XSL-FO	CSS
Box layout	XSL and CSS both generate rectangular boxes by applying styles to markup. Their features are mostly identical.	
Page layout	• Page masters defined in <fo:simple-page-master> FOs • Page masters selected in a predefined sequence using <fo:page-sequence-master> FOs • AH Formatter supports nested <fo:page-sequence> as a child of <fo:flow> • AH Formatter adds <axf:spread-page-master> for defining pages with regions that spread across two pages • "Flow maps" allow content to flow into multiple separate regions on the same page	• Pages defined in `@page` rules • Elements can specify a `@page` rule to use • Adjacent or nested elements with different `@page` selection causes a page break with the change to the new `@page`
Headers & footers	• Four 'outer' regions may be defined for each <fo:simple-page-master> • Default name for each region may be overridden • Block-level content from <fo:static-content> directed to named region (if defined for current page) • Variable content specified with <fo:marker> and retrieved with <fo:retrieve-marker> • Retrieved content does not inherit properties from its original location	• 16 page-margin boxes on every page • Page-margin box names are fixed and each box has predefined default alignment • Content is either retrieved from a running element using `running()` or is any combination of fixed strings, counters, and strings retrieved using `string()` • Running elements inherit properties from their original location
Multiple columns	• Only in <fo:region-body> and, as AH Formatter extension, in <fo:block-container> • `column-count` specifies fixed number of columns • AH Formatter extensions for column balancing and appearance of column rule	• Any block-level element • `column-count` specifies fixed number of columns • Setting `column-width` generates as many columns as will fit • AH Formatter extensions for column balancing and appearance of column rule

Section	XSL-FO	CSS
Keeps & breaks	• Keywords plus numeric values to indicate weight • AH Formatter extension for maximum height for a keep-together condition	• Keywords only • AH Formatter extension for maximum height for a keep-together condition
Paragraph setting	• Same text alignment control, including AH Formatter extensions • AH Formatter extensions for baseline grid • Overflow extensions	• Same text alignment control, including AH Formatter extensions • AH Formatter extensions for baseline grid
Footnotes & sidenotes	• A 'footnote-reference-area' is implicit in the area generated by an <fo:region-body> • Footnote number expected to be included in XSL-FO • Sidenotes are an AH Formatter extension • Sidenote number expected to be included in XSL-FO	• `@footnote` rule for footnote area included in default `html.css` • Footnote numbering is automatic • `@sidenote` rule for sidenote area included in default `html.css` • Sidenote numbering is automatic
Tables	• XSL 1.1 tables based on CSS 2 tables • A table with a caption requires <fo:table-and-caption> containing both <fo:table-caption> and <fo:table> • Precedence of collapsing borders can be set in the XSL-FO • AH Formatter extensions for table footnotes, accessibility, and improved control of breaks	• CSS still uses CSS 2 tables • AH Formatter implements XSL-FO properties for cell content alignment and table header and footer behavior at breaks • AH Formatter extensions for accessibility and improved control of breaks • `-ah-reference-orientation` applies to tables and table cells
Lists	• Multiple FOs for parts of a list • List markers expected to be included in XSL-FO • Numeric markers can be formatted using counter styles or other formats	• Any element may have `display: list-item;` to render as a list item • `::marker` pseudo-element for list item marker • Marker in an ordered list typically generated using a counter
Character setting	Equivalent capabilities. Many properties are common to XSL-FO and CSS. Properties that are not defined in a technology are implemented as AH Formatter extensions, plus there are multiple original AH Formatter extensions implemented for both XSL-FO and CSS.	

Section	XSL-FO	CSS
Japanese text composition	Equivalent capabilities. Many properties are common to XSL-FO and CSS. Properties that are not defined in a technology are implemented as AH Formatter extensions, plus there are multiple original AH Formatter extensions implemented for both XSL-FO and CSS.	
Cross-references	• <fo:basic-link> has `internal-destination` and `external-destination` properties • Only one of `internal-destination` and `external-destination` should be specified • Content of cross-reference, including section number, etc., is expected to be included in XSL-FO • Page number of target generated using <fo:page-number-citation> • <fo:page-number-citation-last> generates page number of last area generated by an FO • AH Formatter extensions for physical page number, page numbers in reverse sequence, and relative page number difference	• `href` used for both internal and external links • Text (but not styling or contained markup) of target can be retrieved using `target-text()` • Number (e.g., section number) and page number can be calculated using `target-counter()` • AH Formatter extensions for physical page number and page numbers in reverse sequence
Image positioning	AH Formatter provides equivalent extensions for both XSL-FO and CSS that provide better capabilities than what is defined in either XSL-FO or CSS.	
MathML & SVG graphics	• MathML and SVG are not part of XSL-FO but can be included using <fo:instream-foreign-object> or referred to using <fo:external-graphic> • AH Formatter provides custom MathML3 and SVG renderers	• MathML and SVG are part of HTML5 • AH Formatter provides custom MathML3 and SVG renderers

Section	XSL-FO	CSS
Counters	• XSL 1.1 has a limited ability to format page numbers using `format` and other properties that are based on the *Number to String Conversion Attributes* defined in XSLT 1.0 20 • All other formatted numbers are expected to be included in the XSL-FO • AH Formatter implements CSS 3 counter styles, including defining custom counter styles in the XSL-FO, to generate numbers when formatting • Counter styles can be used on all block-level and inline-level formatting objects	• Numbers based on the element structure can be generated using `counter()` • Counters can be incremented or reset at arbitrary element starts or at page boundaries using `counter-increment` and `counter-reset` properties • CSS Counter Styles 3 4 defines many formats for presentation of counter values as well as a mechanism for custom counter styles

Color	• RGB and CMYK colors
	• ICC color profiles
	• AH Formatter adds:
	  • RGBA, HSL, HSLA, and CMYKA colors
	  • Extended color names from CSS
	  • Gradient functions from CSS3
		• RGB, RGBA, HSL, HSLA, and CMYK colors
		• CMYKA as AH Formatter extension
		• Extended color names
		• Radial and linear gradients

Borders & background	Equivalent capabilities. Many properties are common to XSL-FO and CSS. XSL-FO and CSS 2 support a single background image. CSS 3 and AH Formatter support multiple background images. Properties that are not defined in a technology are implemented as AH Formatter extensions, plus there are multiple original AH Formatter extensions implemented for both XSL-FO and CSS.	
PDF output	• PDF bookmarks from explicit <fo:bookmark-tree> • Bookmarks can link to anywhere inside or outside the document	• PDF bookmarks from elements with `bookmark-level` property
	AH Formatter extensions include: • Bookmarks can be built from document structure • PDF forms • Multiple PDF variants, including PDF/A, PDF/X, and PDF/UA • XSL-FO can be output as multiple volumes	

Section	XSL-FO	CSS
Indexes	• Multiple FOs specific to indexes • Properties for merging repeated and consecutive page numbers in index entries	• Extension properties for merging repeated and consecutive page numbers can be used on any block-level element

5. Formatting the XSL-FO/CSS comparison

The *XSL-FO/CSS Comparison* document on which this paper is based grew out of AH Formatter customer requests for information about the differences between XSL-FO and CSS. An initial list prepared by Antenna House Support became the table in the previous section, and that has now expanded into a 150-page book.

The book is available in two versions—formatted using XSL-FO and formatted using CSS—for you to compare. However, the results are so similar that for many pages, you will need the Antenna House Regression Testing System 2 to find the differences.

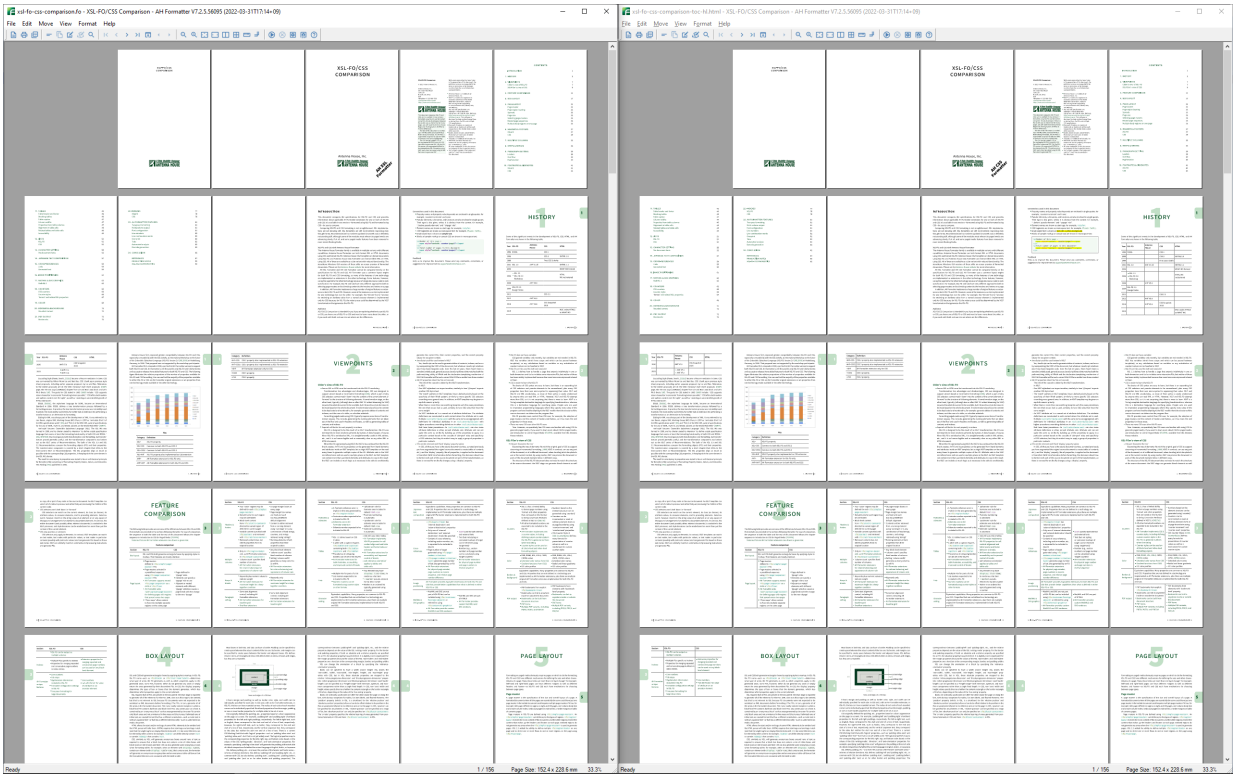


Figure 1. CSS and XSL-FO versions side-by-side

The text is marked up in XHTML5 (XML-serialized HTML5). Before being formatted, the XHTML5 is augmented using XSLT to add the table of contents, side tabs, and syntax highlighting.

The CSS version applies CSS to the augmented XHTML5 using AH CSS Formatter to generate PDF. For the XSL-FO version, the augmented XHTML5 is

transformed into XSL-FO markup using XSLT and is then formatted using AH XSL Formatter to generate PDF.

5.1. Development sequence

For faster development as well as consistent look-and-feel, the book is written in XHTML and uses the same styles as the *Introduction to CSS for Paged Media* 8 book. That book is itself an expansion of the translation of a Japanese book written in 2005. Some of the markup conventions and class names are unchanged from the 2005 predecessor.

The CSS styles had previously only been used for *Introduction to CSS for Paged Media*, so a necessary step was to modularise the processing so that the two documents can use the same core stylesheets with local overrides and local graphics.

The other step was to develop the XSLT stylesheets for transforming the XHTML into XSL-FO. Instead of repeating the logic that augments the source XHTML to generate the XHTML that is formatted using CSS, the stylesheets for generating XSL-FO use the augmented XHTML as their source.

To further reduce development time, the custom XSL-FO stylesheets import an XSLT 3 version of the XHTML to XSL-FO stylesheets 15 by Antenna House.

5.2. CSS version

As stated previously, the version formatted using CSS uses existing XSLT and CSS stylesheets.

5.2.1. Alternative approaches

Inasmuch as the XHTML is written to use the pre-existing XSLT and CSS stylesheets, it would not make sense to revise the CSS for this book.

5.3. XSL-FO version

5.3.1. Page layout

Page dimensions in CSS are defined in `@page` rules. Individual elements can specify which `@page` rule to use. If that is not the current `@page` rule, the formatter will start a new page using the new `@page` rule. In contrast, XSL provides <fo:simple-page-master> for defining the dimensions of a page and its constituent regions plus <fo:page-sequence-master> for defining which page master to use in the progression of pages for an <fo:page-sequence>.

Because of these differences between CSS and XSL-FO, the XSL-FO page masters were written based on the CSS originals, but they are also parameterised through the use of variables and attribute sets.

```
<xsl:template name="make-layout-master-set">
  <fo:layout-master-set>
    <xsl:call-template name="page-master-title" />
    <xsl:call-template name="page-master-chapter" />
    <xsl:call-template name="page-master-chapter">
      <xsl:with-param name="name" select="'chapter-two-column'"
                      as="xs:string" />
      <xsl:with-param name="column-count" select="2" as="xs:integer" />
    </xsl:call-template>
    <xsl:call-template name="page-master-landscape-wide" />
    <xsl:call-template name="page-master-blurb" />
  </fo:layout-master-set>
</xsl:template>

<xsl:template name="page-master-chapter">
  <xsl:param name="name" select="'chapter'"
             as="xs:string" />
  <xsl:param name="column-count" select="1" as="xs:integer" />

  <fo:simple-page-master master-name="{$name}-first"
                         xsl:use-attribute-sets="page bleed-crop-right">
    <fo:region-body margin-top="{$page-margin-top}"
                    margin-right="{$page-margin-outside}"
                    margin-bottom="{$page-margin-bottom}"
                    margin-left="{$page-margin-inside}"
                    column-count="{$column-count}" />
    <fo:region-after region-name="right-footer"
                     text-align="right"
                     display-align="center"
                     extent="{$page-margin-bottom}" />
    <fo:region-end region-name="first-tab"
                   extent="{$page-margin-outside}" />
  </fo:simple-page-master>
  <fo:simple-page-master master-name="{$name}-left"
                         xsl:use-attribute-sets="page bleed-crop-left">
    <fo:region-body margin-top="{$page-margin-top}"
                    margin-right="{$page-margin-inside}"
                    margin-bottom="{$page-margin-bottom}"
                    margin-left="{$page-margin-outside}"
                    column-count="{$column-count}" />
    <fo:region-after region-name="left-footer"
                     text-align="left"
                     display-align="center"
                     extent="{$page-margin-bottom}" />
    <fo:region-start region-name="left-tab"
                     extent="{$page-margin-outside}" />
```

```
    </fo:simple-page-master>
...
    <fo:page-sequence-master master-name="{$name}">
      <fo:single-page-master-reference master-reference="{$name}-first" />
      <fo:repeatable-page-master-alternatives>
        <fo:conditional-page-master-reference
            master-reference="{$name}-blank-left"
            blank-or-not-blank="blank"
            odd-or-even="even" />
        <fo:conditional-page-master-reference
            master-reference="{$name}-blank-right"
            blank-or-not-blank="blank"
            odd-or-even="odd" />
        <fo:conditional-page-master-reference
            master-reference="{$name}-left"
            odd-or-even="even" />
        <fo:conditional-page-master-reference
            master-reference="{$name}-right"
            odd-or-even="odd" />
      </fo:repeatable-page-master-alternatives>
    </fo:page-sequence-master>
  </xsl:template>
```

5.3.2. Side tabs

The formatted book features chapter numbers in side tabs, or index tabs, on the outer edges of pages. They are a navigation aid, although they were included in the original *Introduction to CSS for Paged Media* largely to illustrate how they can be done.

CSS does not allow calculations based on the position of an element, so the XSLT that augments the XHTML source adds a <p> that contains the chapter number to be taken out of the flow and used in the side tab. The element also has a data-position attribute that is used when calculating the vertical offset of the side tab.

The XSLT that augments the source XHTML calculates the position based on the allowed number of tabs per page:

```
  <xsl:template match="h2">
    <xsl:copy>
      <xsl:apply-templates select="@*" />
      <xsl:call-template name="data-bookmark" />
      <xsl:if test="empty(@id)">
        <xsl:attribute name="id" select="ahf:generate-id(.)" />
      </xsl:if>
      <xsl:apply-templates select="node()" />
    </xsl:copy>
```

```
  <xsl:if test="ahf:is-numbered(.)">
    <xsl:variable name="number">
      <xsl:number count="h2[ahf:is-numbered(.)]" level="any" format="1"/>
    </xsl:variable>
    <p class="tab" data-position="{($number - 1 ) mod $tabs-per-page}">
      <xsl:value-of select="$number" />
    </p>
    <p class="tab first" data-position="{($number - 1 ) mod $tabs-per-
page}">
      <xsl:value-of select="$number" />
    </p>
  </xsl:if>
</xsl:template>
```

The CSS removes the tab numbers from the flow and uses them as running elements:

```
@media ah-formatter {
    p.tab {
        height: 53pt;
        position: running(Tab);
        margin-top: calc(attr(data-position) * 53pt);
        font-variant: proportional-nums;
    }

    p.tab.first {
        position: running(TabChapter);
        color: var(--ah-green-rgb);
        color: var(--ah-green-cmyk);
    }
}
```

The running elements are used in the `@left-top` or `@right-top` page-margin box on the first or subsequent pages of each chapter:

```
@page Chapter:left {
  @left-top {
    content: element(Tab);
  }
}

@page Chapter:right {
  @right-top {
    content: element(Tab);
  }
}

@page Chapter:first {
```

```
    @right-top { content: element(TabChapter) }
  }
```

When generating the XSL-FO, the tab number and position in the XHTML are reused, rather than repeating the logic to generate the same numbers again. The tabs are generated as the content of an <fo:static-content> that is directed to either the <fo:region-start> or <fo:region-end> of a page:

```
<!-- Tabs only for chapters. -->
<xsl:if test="tokenize(@class, '\s+') = 'Chapter'">
  <fo:static-content flow-name="first-tab">
    <fo:block-container color="{$ah-green-cmyk}"
                        xsl:use-attribute-sets="tab">
      <fo:block-container margin="0">
        <fo:block>
          <xsl:value-of select="p[@class = 'tab']" />
        </fo:block>
      </fo:block-container>
    </fo:block-container>
  </fo:static-content>
  <fo:static-content flow-name="left-tab">
    <fo:block-container xsl:use-attribute-sets="tab"
                        margin-left="-{$tab-width} div 2">
      <fo:block-container margin="0">
        <fo:block>
          <xsl:value-of select="p[@class = 'tab']" />
        </fo:block>
      </fo:block-container>
    </fo:block-container>
  </fo:static-content>
  <fo:static-content flow-name="right-tab">
    <fo:block-container xsl:use-attribute-sets="tab">
      <fo:block-container margin="0">
        <fo:block>
          <xsl:value-of select="p[@class = 'tab']" />
        </fo:block>
      </fo:block-container>
    </fo:block-container>
  </fo:static-content>
```

Because the attributes in an attribute set are evaluated each time they are used, the `margin-top` is calculated for each tab:

```
<xsl:attribute-set name="tab">
  <xsl:attribute name="color" select="'white'" />
  ...
  <!-- Generate an expression for the formatter to evaluate.
       Attributes in an attribute set are reevaluated every time that
```

```
        the attribute set is used. -->
  <xsl:attribute name="margin-top"
                 select="concat($page-margin-top,
                                ' + ',
                                $tab-height,
                                ' * ',
                                p[@class = 'tab']/@data-position)" />
  <xsl:attribute name="margin-left" select="'4.5mm'" />
  <xsl:attribute name="margin-right" select="'4.5mm'" />
</xsl:attribute-set>
```

5.3.3. Tables

The imported `xhtml2fo.xsl` stylesheet makes extensive use of `xsl:attribute-set`, reflecting that the original version was written in XSLT 1.0. The XSLT 3.0 stylesheet also supports passing a map of property overrides when formatting, for example, tables and lists.

Most but not all tables in the XHTML document are styled using the `StdTable` styles. This is straightforward with CSS, where the class name is part of the CSS selector and the CSS specificity rules ensure that the rules with `.StdTable` override the default table styles. For example:

```
table.StdTable {
  border-color: gray;
  border-style: solid none solid none;
  border-width: 1.5pt;
  border-collapse: collapse;
  margin-top: 1em;
  margin-bottom: 1em;
  margin-left: auto;
  margin-right: auto;
}
```

It is less straightforward when using XSLT to generate XSL-FO because it is harder to override some XSL-FO properties but not others while using a generic template to process all table elements. The map of properties approach passes specific property values to the generic templates for the table-related properties:

```
<xsl:template match="table[tokenize(@class, '\s+') = 'StdTable']">
  <xsl:param name="table-properties"
             select="map { }"
             tunnel="yes"
             as="map(xs:string, map(xs:string, xs:string?))" />

  <xsl:variable
      name="local-table-properties"
      select="map {
```

```
'table-caption' : map { 'font-weight' : 'bold',
                        'margin-top' : '0',
                        'margin' : '0.5em' },
'table' : map { 'border-collapse' : 'collapse-with-precedence',
                (: CSS puts the top border inside the table, but
                   XSL-FO puts half the border above the table. :)
                'margin-top' : '0.75pt',
                'margin-left' : 'auto',
                'margin-right' : 'auto',
                'border-top' : '1.5pt solid gray',
                'border-bottom' : '1.5pt solid gray',
                'border-left-style' : 'hidden',
                'border-right-style' : 'hidden' },
'table-header' : map { 'border-style' : 'solid none',
                       'border-color' : 'gray transparent',
                       'border-width' : '1.5pt',
                       'border-after-precedence' : 'force' },
'table-cell' : map { 'border-style' : 'solid',
                     'border-color' : 'gray',
                     'border-width' : '0.75pt',
                     'padding' : '3pt',
                     'text-align' : 'left' }
}"
      as="map(xs:string, map(xs:string, xs:string?))" />

  <!-- Properties from importing stylesheet or higher-priority
       template have precedence over local properties. -->
  <xsl:variable
      name="table-properties"
      select="map:put($table-properties,
                      'table-and-caption',
                      map:merge(($table-properties('table-and-caption'),
                                 $local-table-properties('table-and-
caption'))))"
      as="map(xs:string, map(xs:string, xs:string?))" />
  <xsl:variable
      name="table-properties"
      select="map:put($table-properties,
                      'table-caption',
                      map:merge(($table-properties('table-caption'),
                                 $local-table-properties('table-
caption'))))"
      as="map(xs:string, map(xs:string, xs:string?))" />
  <xsl:variable
      name="table-properties"
      select="map:put($table-properties,
```

```
                    'table',
                    map:merge(($table-properties('table'),
                              $local-table-properties('table'))))"
      as="map(xs:string, map(xs:string, xs:string?))" />
  <xsl:variable
      name="table-properties"
      select="map:put($table-properties,
                    'table-header',
                    map:merge(($table-properties('table-header'),
                              $local-table-properties('table-
header'))))"
      as="map(xs:string, map(xs:string, xs:string?))" />
  <xsl:variable
      name="table-properties"
      select="map:put($table-properties,
                    'table-cell',
                    map:merge(($table-properties('table-cell'),
                              $local-table-properties('table-
cell'))))"
      as="map(xs:string, map(xs:string, xs:string?))" />

  <xsl:next-match>
    <xsl:with-param
        name="table-properties"
        select="$table-properties"
        tunnel="yes"
        as="map(xs:string, map(xs:string, xs:string?))" />
  </xsl:next-match>
</xsl:template>
...
<xsl:template match="thead">
  <xsl:param name="table-properties"
             select="map { }"
             tunnel="yes"
             as="map(xs:string, map(xs:string, xs:string?))" />

  <fo:table-header xsl:use-attribute-sets="thead">
      <xsl:sequence
          select="ahf:add-properties($table-properties('table-header'),
                            $border-properties)" />
      <xsl:call-template name="process-table-rowgroup">
        <xsl:with-param name="properties"
                        select="$table-properties('table-header')"
                        tunnel="yes" as="map(xs:string, xs:string?)?" />
     </xsl:call-template>
```

```
    </fo:table-header>
  </xsl:template>
```

5.3.4. Alternative approaches

Because the XSLT stylesheets to generate the XSL-FO are written by hand, the XHTML source currently avoids using any style attributes. In principle, the manual stylesheet creation could be replaced by an automated system that processes the XHTML document together with its stylesheets using the 'css-tools' utility 22 by Gerrit Imsieke and le-tex publishing services. 'css-tools' resolves the cascade of CSS styles, including from style attributes, and generates applicable CSS properties as separate XML attributes on each element. 22 The structure of the XHTML document, in combination with the XML attributes representing the styles, could be used to generate FOs and properties for an XSL-FO equivalent of the XHTML +CSS source. In practice, however, while 'css-tools' has recently improved its handling of CSS pseudo-elements, it does not yet handle `@page` rules or custom `@media` types.

6. Conclusion

XSL-FO and CSS have a lot of similarities because of their commitments to use common properties when XSL 1.0 and CSS2 were developed. More differences developed as XSL and CSS each added new features. AH Formatter smooths out most of the differences by providing many of the properties from one technology as extensions available to the other technology.

Bibliography

[1] Bert Bos. *Report on the W3C style sheet workshop, Paris '95*. https://www.w3.org/Style/951106_Workshop/report1.html

[2] Antenna House, Inc. *Antenna House Regression Testing System*. https://www.antennahouse.com/ahrts

[3] World Wide Web Consortium. *Cascading Style Sheets, level 1*. W3C Recommendation 17 December 1996.https://www.w3.org/TR/REC-CSS1/

[4] World Wide Web Consortium. *CSS Counter Styles Level 3*, W3C Candidate Recommendation, 14 December 2017. https://www.w3.org/TR/2017/CR-css-counter-styles-3-20171214/

[5] Wikipedia. *Document Style Semantics and Specification Language*. https://en.wikipedia.org/wiki/Document_Style_Semantics_and_Specification_Language

[6] World Wide Web Consortium. *CSS3 module: Generated Content for Paged Media*. 4 May 2007. http://www.w3.org/TR/2007/WD-css3-gcpm-20070504

[7] World Wide Web Consortium. *Historical Style Sheet proposals*. 6 January 2021. https://www.w3.org/Style/History/ (archive[1])

[8] Antenna House. *Introduction to CSS for Paged Media*, 15 February 2019. https://www.antennahouse.com/css

[9] World Wide Web Consortium. *Requirements for Japanese Text Layout 日本語組版処理の要件（日本語版）*. 11 August 2020. URL: *https://www.w3.org/TR/jlreq/*

[10] Håkon Wium Lie. *Cascading Style Sheets*. Ph.D Thesis. 2005. https://wiumlie.no/2006/phd/ (archive[2])

[11] Håkon Wium Lie and Bert Bos. *The CSS saga*. https://www.w3.org/Style/LieBos2e/history/ (archive[3])

[12] World Wide Web Consortium. *Consistency of Formatting Property Names, Values, and Semantics*. TAG Finding. 25 July 2002. https://www.w3.org/2001/tag/doc/formatting-properties.html

[13] World Wide Web Consortium. *Caption & Summary* in *WAI Web Accessibility Tutorials*. 27 July 2019. https://www.w3.org/WAI/tutorials/tables/caption-summary/

[14] WHAT-WG. *The CSS user agent style sheet and presentational hints*. 12 August 2021. https://html.spec.whatwg.org/multipage/rendering.html#the-css-user-agent-style-sheet-and-presentational-hints

[15] Antenna House, Inc. *XHTML to XSL-FO* in *Developing XSL-FO Stylesheets*. https://www.antennahouse.com/xml-to-xsl-fo-stylesheets

[16] World Wide Web Consortium. *Extensible Stylesheet Language (XSL)*. W3C Recommendation. https://www.w3.org/TR/2001/REC-xsl-20011015/

[17] Jon Bosak. *XS discussion begins*. 22 May 1997. http://xml.coverpages.org/xs-970524.html (archive[4])

[18] World Wide Web Consortium. *Report from International Workshop on the future of the Extensible Stylesheet Language (XSL-FO) Version 2.0*. 18 October 2006. https://www.w3.org/Style/XSL/2006-Workshop/Report.html

[19] World Wide Web Consortium. *Charter - XSL Working Group*. 22 February 2002. https://www.w3.org/Style/2000/xsl-charter.html

[1] http://web.archive.org/web/20201031230541/https://www.w3.org/Style/History/
[2] http://web.archive.org/web/20201130075146/https://wiumlie.no/2006/phd/
[3] http://web.archive.org/web/20201018013706/https://www.w3.org/Style/LieBos2e/history/
[4] http://web.archive.org/web/20200112005134/http://xml.coverpages.org/xs-970524.html

[20] World Wide Web Consortium. *XSL Transformations (XSLT)*. W3C Recommendation. 16 November 1999. https://www.w3.org/TR/1999/REC-xslt-19991116

[21] Jon Bosak. *XML Part 3: Style [NOT YET] Version 1.0*. http://sunsite.unc.edu/pub/sun-info/standards/dsssl/xs/xs970522.rtf.zip (archive[5])

[22] le-tex publishing services, *css:expand*, https://github.com/transpect/css-tools

[23] Gerrit Imisieke, *Conveying Layout Information with CSSa*, XML Prague 2013, https://archive.xmlprague.cz/2013/presentations/Conveying_Layout_Information_with_CSSa/CSSa_xmlprague_gimsieke.html#/step-1

[24] World Wide Web Consortium, *Cascading Style Sheets (CSS) Working Group Charter*, 2014, https://www.w3.org/Style/2014/css-charter

[25] World Wide Web Consortium, *CSS Working Group Charter*, 2016, https://www.w3.org/Style/2016/css-2016.html

[5] http://web.archive.org/web/20200112005134/http://sunsite.unc.edu/pub/sun-info/standards/dsssl/xs/xs970522.rtf.zip

Structure! You get more than you see

Cerstin Mahlow
Zurich University of Applied Sciences, School of Applied Linguistics
<cerstin.mahlow@zhaw.ch>

Abstract

In the 1990s, the focus on the printed page as the final product of writing with WYSIWYG tools clashed first with the development of the Web and a decade later with the advent of mobile devices. Both developments enabled—and required—new types of documents and thus demanded new tools and processes for producing these documents. In the 2010s, the emphasis on writing experience, personalization of tools, and the growing diversity of input devices, methods, and displays is the main reason for the design and development of "new writing tools." Their functionalities are often working implementations of methods and concepts originally described and developed in the 1960s and 1970s that seem to have failed due to the limitations of computers at that time. Dedicated research on writing tools stopped in the late 1980s, once universities and companies had decided what to purchase and Microsoft Word had achieved monopoly status in the consumer market. The shift of academic writing to include dynamic aspects of "text," e.g., code (snippets), data plots, and other visualizations clearly demands other tools for text production than traditional word processors. When the printed page no longer is the desired final product, content and format can be addressed explicitely and separately, thus emphasizing the structure of texts rather than the structure of documents.

1. Introduction

In the early 1990s, Microsoft Word became the de facto market leader in word processing software for personal use. The main reason was that MS Word was bundled with many PCs. From a customer's perspective, everything was ready: one could just turn on the new computer and start writing with MS Word. Getting and installing a different word processor like WordPerfect would have involved purchasing another license and installing another program. This ubiquity lead to several effects: writers became accustomed to the appearance, features, and affordances of MS Word; the format of text files produced with MS Word became the default file format expected and demanded for submissions of academic texts and beyond, for interchange between writers when writing collaboratively, and for further processing in publishing houses.

However, also in the 1990s, the focus on the printed page as the final product of writing with WYSIWYG tools clashed first with the development of the Web and a decade later with the advent of mobile devices. Both developments enabled —and required—new types of documents and thus demanded new tools and processes for producing these documents.

Thirty years later we face a wave of tools for text production without any explicit document format (understood as printable object) as final product. Writers of all kinds are invited and seem to enjoy a "new" writing experience focusing on content and some very basic textual structures and decorations.

In this paper I will first look at the production of documents from a writer's perspective. Clearly, the intended use and final distribution format influences tools and procedures involved in document creation. I argue that both a general shift in what we consider "final" documents due to general technical progress, as well as customers demanding tools to be tailored and configurable for specific needs—which also, due to general technical progress, today can be satisfied rather easily compared to 20 or 40 years ago—fosters a movement towards new tools. I will show that the underlying ideas are not completely new, but today's technology finally allows for the actual implementation of ground-breaking concepts from the earlier decades since the 1960s. We are witnessing a trend towards explicitely addressing and working on format and content separately. In the last section, I propose to apply a different view on structure with respect to natural language documents: When we get rid of the concept of the printed page as the main instantiation of texts, we actually work on *text* strucutures, not on *document* structures—which has all kinds of impacts on use and development of tools for writing.

2. Documents

2.1. Focus on the standard-format printed page

For a long time, documents mainly have been spread and consumed as physical pages, i.e., "the document" as the final product of writing was a written or printed page as a physical object. With the advent of computers to be used for writing, this was initially achieved by using structured-editing with troff [27], for example. Troff files would make structure explicit, but the target document could only be seen when printed later. Almost nobody can interpret troff commands mentally, imagine the final product, and then decide what to change where. Various studies showed that writers missed an overview or a "global perspective" [37] on the text as a whole for general orientation and also for revision decisions when working with early word processors. [19] [22]

Printing for checking the current look of an intermediate draft is expensive and complicated. There originated the demand to see on the screen as well as to

be able to write and edit a document in a rendering pretty close to the final product. Writers wanted to work on their text document while seeing it as it would appear on paper when actually printed.

2.2. WYSIWYG

The slogan "What You See Is What You Get" (WYSIWYG), first used for advertising WordStar, referred to "seeing on the screen" the page as it would be printed later. Although it was Easy Writer which was the first word processor that allowed text to be displayed on the screen exactly as it would later be printed. [3] The emphasis was on the printed page and on how to best support writers to lay out their text on that page. WYSIWYG word processors allowed writers to write text and manipulate the appearance of that text at the same time. Word processors started to include features to typeset text into documents. As developers tried to integrate all aspects of the creation of paper documents into a single tool, the number of features and menu options of word processors like MS Word grew without bounds.

Writers took over the roles of typesetters and layouters. Working on content and form—i.e., layout—at the same time became possible. Apart from traditional possibilities to emphasize text as bold, italic, or underlined, the writer could adjust the font, its color and size. Writers could also use format templates for documents which render the text according to its stucture and defined appearances for headings, lists, etc.

WYSIWYG applications often tempt writers to configure appearance of the text by applying formatting directly, which might result in the desired look, but is error prone and will cause inconsistencies. In various studies with early WYSIWYG word processors [33], [37], and [13] found that they hampered and supported revising and editing at the same time: the document always looks somehow "finished" but revising is much easier than when using pen and paper or a typewriter and leaves no traces. Even 10 years earlier, [8] notes:

> *The text editor also eliminates the spatial and aesthetic barriers that are special inhibitators of revising activity. Writers are often reluctant to mess up a carefully written page by crossing out words or cramping inserts between lines and in the margins." [8]*

2.3. Dynamic Documents

In the 1990s the development of the Web enabled and demanded other documents, not intended to be distributed as printed pages. It allows for *dynamic* documents with respect to form and content. One major aspect of the Web is the linking of documents as hypertext, which challenges authors during writing. The understanding of "text" changed at the turn of the century to include "interactive,

hypertextual documents—many of which reside on the Internet—[which] use color, sound, images, video, words, and icons to express their messages" [17]; this clearly required tools that would allow writers to create and edit such documents.

Today's websites are dynamic, they are configurable to preferences of visitors, multi-media, assemblages of other dynamic modules, they collect user data, etc. Software originally developed to support writing of personal diary-style texts—weblogs/blogs—now include automatic rendering of the articles (text as well as static and dynamic images or sound) for all kinds of displays and devices. General markup for structuring text into headers, paragraphs, for emphasis, and linking draws on what is used elsewhere, for example in Wikipedia, and respective editors for authoring Wikipedia-style articles have been developed. As the general idea of Wikipedia as collection of linked texts became popular, this principle gained ground.

What we also see with wikis is the return of the explicit separation of content and form—what one sees during writing and editing is *not* what one sees when looking at the result of these activities. Whether or not users find it comfortable to have a constantly updated preview (side by side with the editing field/window or in a separate window/application) is up to them and can be arranged as well as switched off and on.

2.4. Single-source publishing

Taking into account that communication takes place on various channels with specific and complex formats emphasizes structure within *text,* which allows the display of the content/text according to features of devices and tailored to the needs of readers. Writing in these scenarios used to be challenging and required knowledge of specific markup to be used for rendering.

Format templates to be configured and applied in the word processor itself have been an attempt to distinguish form and content for the general writer. They allow for formating documents consistently according to specific rules that apply to structural elements and might depend on the desired output format in terms of document size and file format. For example, rendering of headings or URLs will differ for a PDF document to be printed and for a HTML document to be displayed and used interactively in a web browser.

The idea of "single-souce publishing" addressed varying distribution formats and channels but allowed the writer to only produce the text once and then leave rendering to somebody or something else. While WYSIWYG word processors mix content and form and allow users to work on both nearly at the same time, there is an ever growing trend to explicitly split these two aspects again.

3. Writing Tools

Even when the "paperless office" was advertised more and more, the form in which a text was distributed and consumed more often than not was still a printed page—or the PDF file as a simulacrum of paper and thus a print-oriented document. Many tools were developed around this central idea: writing technology became digitalized, but no "digital transformation" seemed to be around the corner. As I showed above, the assumption of a print-oriented page as final document is not that strong any longer, though.

Digitalization uses digitized objects, which still refer to their original features and affordances [18]. They are intended to be manipulated with digital tools, which often also still refer to corresponding real-world tools, their features, and their affordances. After the advent of graphical user interfaces, the design of word processors mimicked for a long time the look (and feel) and functionality of typewriters.

3.1. Early word processing tools

The term "word processing" was first used in the 1960s. It did not refer to a single software application as today, but to a complete solution consisting of hardware and software, the market leader at that time was IBM. (For historical overviews see [49]; [20]; [14]) Existing technologies for storing, indexing, searching and finding texts were combined and extended by functions known from editors for manipulating program code and data—i.e., text editors.

However, this idea only caught on and became commercially attractive in the early 1990s. Before that—in the mid-1970s—programs such as Electric Pencil or Easy Writer were developed but were later unable to hold their own against commercial products. [3] Similarly, word processors developed in the late 1980s by research institutions and universities did not succeed as they required mainframes while commercial products were much cheaper and ran on PCs. [32] [44] The first commercially successful word processing programs like WordStar or later WordPerfect disappeared, mainly due to decisions and advertising strategies of the management of the respective companies, not due to features or writers' preferences. [3] [4] [14] [49]

The "old" word processing programs were forgotten, which is regrettable if one compares functions available today in word processing programs with the possibilities of editors from the 1960s or 1970s.

As an example, consider NLS (oN-Line System) [15], the Stanford Research Institute editor famously introduced by Douglas C. Engelbart in 1968.[1] NLS already worked with multiple windows in 1968 and could be operated with the

[1] This performance is considered the "mother of all demos." Among other things, the screen of Engelbart's computer was transmitted live to a large video screen.

mouse—another invention of Engelbart. NLS allowed both text writing and data exchange, use of a shared database, e-mail, and simultaneous editing of a document (called "computer conferencing" [5] even when writers were physically separated). It included word wrap, search and replace, cut and paste between documents, etc. [20] Only some years later, [12] developed the Hypertext Editing System (HES).

3.2. The default word processor sets the bar and stops development

The accustomization of writers to MS Word and its perception as the default word processor resulted in a general assumption that every other new writing and editing facility—e.g., in the first learning management systems, which started to appear in the early 2000s—should be designed to resemble the look and feel of MS Word and include its main features to offer a familiar user experience.

The design and implementation of Google Docs as well as what is today LibreOffice are further strong examples: Google Docs set off based on Writely, a web-based collaborative word processor supporting concurrent editing developed by the company Upstartle to "become the blog-posting tool of choice and be the fastest, easiest and most widely-adopted way to make a wiki, create a document, collaborate, or post a web page" [7]. Google itself supported the development of MobWrite, a web-based open-source multi-user real-time plain-text editor [16] addressing the issue of keeping a consistent synchronized document during concurrent editing by various users. Documents could be saved in the standard MS Word format, thus ensuring interoperability. In contrast to Google Docs, the current web version of MS Word is not free of charge, and only in 2021 Microsoft began offering a full version of MS Word as a web version, hoping to keep up with their more popular competitor Google.

Writer, the word processor in LibreOffice—a fork from OpenOffice, which itself is an open-sourced version of StarOffice—,by default saves documents in the OpenDocument format, but it can also export to various other formats including XHTML, RTF, and MS Word's `.doc` and `.docx` to allow for subsequent editing using other word processors. Features and menues resemble MS Word, as Google Docs does. All of them are still WYSIWYG word processors with the print-oriented page as the goal for "what to get." It once set out as marketing slogan but seems to be the standard and default requirement for word processing software for quite some time now. However, Google Docs now also allows settings for "pageless" documents.

3.3. Scientific research on writing tools

Scientific Research on writing tools more or less stopped in the late 1980s[2], when universities and companies increasingly settled on "standard" software (for large-scale installations) (e.g., [8]; [42]; [21]; [35]) and MS Word had achieved monopoly

status in the consumer market. In the early 1980s [36] and [45] had explored "How do people really use text editors?" (title of ([45])) to draw conclusions for future developments of editors with respect to functionality and general design. Ten years later, [23] stated:

> *It seems clear however that in order to produce computer based tools to support writers and the writing process we must increase our knowledge of how writers conduct their craft. An increased understanding of writer's requirements and the task involved in writing will form the basis of the next generation of writing tools. [23]*

They made this statement after the failure of a large project on writing support.

Projects like RUSKIN ([48]; [46]), Writer's Assistant ([41]; [40]), Intelligent Workstation ([25]), and Editor's Assistant ([9]; [11]) did not result in marketable products. Aimed at supporting writers for (post-)editing and revision based on linguistic principles and using language resources, the design and development either did not take into account actual user needs (RUSKIN), or the required resources from natural language processing (NLP) were not mature enough at that time to be used in real-world applications. The computing power of PCs in the early 1990s were insufficient for real-time analysis and generation. These factors led to applications that were too limited for practical use (Intelligent Workstation and Editor's Assistant). The integration of NLP technology into word processors beyond grammar checking has been a research topic since the 1980s (e.g., [25]; [26]; [10]; [31]), but did not result in commercial products at that time. Today's computing power as well as maturity of NLP resources allow for another try in this direction. Since 2013, iAWriter as a commercial product offers information functions using NLP techniques to highlight specific aspects of the evolving text, what is typically called "syntax highlighting" to specifically highlight nouns, verbs, adjectives, etc. It is advertised as "using parts of speech to improve your writing"[3] with explicit references that writers deserve the same professional support as programmers enjoy.

Although writing research does study writing in practical settings and works on writing pedagogy, the writing research community still shows only little interest in improving writing *tools* or the development of new ones. Only occasionally the influence of the writing tool and medium are acknowledged ([38]; [6]; [30]).

30 years ago, [47] stated that professional writers, including academics and journalists, seemed to be satisfied with the tools available at that time, i.e., in the early 1990s. They had adapted to these tools and did not seem to be aware that there might be other options. In the early 2000s, only writers who had used

[2] Note that in contrast, research on text editors for programmers both with respect to general features and more design oriented user interface issues is still ongoing and did *not* experience a serious drop.
[3] https://ia.net/writer/support/writing-tips/parts-of-speech

WordPerfect or other word processors "back in the days" sometimes complained about missing functionality in current word processors.

A decade after the Web, in the early 2000s, mobile devices appeared. Mobile devices and the size of their screens clearly influence the size of the final product as well as the possibilities for inputing and editing of text. In the 2010s, the emphasis on writing experience, personalization of tools, and the growing diversity of input devices, input methods, and displays triggered the design and development of truly "new writing tools." However, text entry beyond typing on physical keyboards seems to be interesting mainly from a design point of view. Focusing on affordances, usability, and habits, [29] propose a thumb-based keyboard to support sight-free and one-handed text entry. [24] work on modeling coordination of eye and finger movements to simulate human-like text entry on touchscreens based on artificial intelligence. Their model and simulation serves as basis for future development of touchscreen keyboarding designs.

[28] state that there seem to be a clear separation from research on tools and their features—hypertext authoring in that case—and the authoring process itself. Their work on these issues hopefully stimulates further research.

4. WYSIWYG is dead, long live structure!

Despite its name, the system "Paper" [37] did not adress the look and feel of the final document, but was intended to help writers gain a general global perspective supported by switching between a view on the text and on the logical structure. The metaphor of paper was also used to offer a familiar navigation through the growing text, explicitely making connections to the mental model the writer has or develops with respect to the text being produced. Paper never became an actual product used widely.

Of course the mental representation is highly influenced by the objects and activities the writer is engaged with on a daily basis—paper—and the desired final output—a set of printed pages. As both the everyday surrounding and the format of the final product change, the mental representation and thus the need and demand for support using features of this representation will change. Writers trained to use WYSIWYG word processors *and* having experienced the paper-centered world on document engineering by heart are vanishing. The next generations already grow up without physical keyboards and without paper—they have no connection to principles like "save," "download," or "print."

4.1. General development

As in the first cycle of the development of writing tools in the 1960s and 1970s ([14]; [20]), we now see again the adoption of tools originally intended for programmers to be used for writing all kinds of texts beyond code. These services are

originally designed to support sophisticated version control, documentation, and exchange. As services like GitHub gain popularity also among non-programmers, we see another type of websites hosted and maintained on services and tools with appropriate affordances but not originally intended to be used this way. Only basic markup is available in those editors, which is automatically rendered into aesthetically attractive websites and printable documents (if needed).

The example of GitHub and applications advertised as GitHub-style editors show that users explore tools and use them according to their needs, not only according to the original purposes of these tools. The easier the use of new tools gets, the more we are willing to also give them a try for established tasks. In writing this means: classic document types as journal articles, book chapters, seminar papers, letters, etc. are defined by features of the final *product*. If there are new tools available that still ensure the production of those documents at the very end —including printing on A4 paper—it might be possible to use those new web publishing tools as writing applications. Rendering of text (and accompanying audio, video, and images) is taken care of by dedicated services and processing pipelines. We witness a clear separation of content and form with an implicit instantiation of single-source publishing as various rendering services can be configured to not only typeset and layout a document but also to include or exclude specific parts automatically.

The markup most commonly used in new editing applications is *Markdown*. Markdown only permits users to annotate a limited amount of decoration—e.g., italic, bold, underline—and of document structure elements—e.g., headers, paragraphs. Although publishing houses today mostly use XML technology internally to produce journals and books, the format demanded from authors is typically still an MS Word `.docx` or even `.doc` file. In the 1990s, this requirement for manuscripts effectively forced authors to actually use MS Word, as it was the sole program to reliably produce a file in the desired format. Today, authors are free to write and revise with their personally preferred tool, which might offer an export of the resulting text in MS Word format. Alternatively, those tools may be able to export the document in some XML format or Markdown, which then can be converted into a file in MS Word format by a program like Pandoc[4]. Texts in XML format also can be rendered into paged media by applying Cascading Style Sheets (CSS) in applications like Antenna House Formatter [1].

4.2. Get rid of the concept of “page” in general

Preparing a traditional apparatus of publishing and libraries still relies on paginated formats for cataloguing and referencing, even with all other processing

[4] Pandoc [https:// pandoc.org] is advertised as “Swiss-army knife” for converting files from one markup format into another and back. This also includes taking care of additional files containing bibliographic information and handling citations and references according to the specified style.

steps being entirely digital. This will probably cause a strong influence for some time to come and require pagination of texts. However, it does not require actual printing of physical pages. When even academic publishing by well-established publishing houses in traditional journals moves away from physical pages—the printed journal is no longer the default distributional channel—as well as digital variants—PDFs mirorring paper pages in letter format or as A4—it is probably high time to abandon the concept of "pages" in general. We already see effects as Google Docs not only allows to configure size and appearance of the page as final version of text production, but also settings for a "pageless" document.

Getting rid of the concept of pages leaves the issue of how to reference parts of a text when there are no page numbers? Referencing sections and their headings might be a solution, but depends on the text structure and the rendering of the document. But also here we could make use of ideas from the 1960s such as a the statement numbers used by NLS.

However, even in 2021, Beat Singer (the developer of the RSL hypermedia metamodel and the interactive paper platform iPaper) stated in an interview: "[M]ost of today's digital document formats are 'simulating paper' based on the WYSIWYG … principle and are therefore not fully embracing the new opportunities offered by digital media." [2]

If we accept that there is no one single final form of a document, we could abandon specific rendering for interchanging documents or drafts of documents but distribute content with some very basic markup and a simple proposal including working instructions on how to render this content as a document. The question thus remains: how to produces these things, what to use to write, revise and edit content, what to use to propose and apply exemplary rendering. To answer these questions, inter- and transdiciplinary research is needed in fields like software engineering, document engineering, human-computer-interaction, writing research and writing pedagogy to address inputting, editing, and processing of natural language texts intended to be distributed as documents and read by humans.

5. New Writing Tools: Back to the Future

In some respects, we now see a development back to ideas and applications of the late 1960s, before projects like NLS where "pushed aside in favor of computer systems more oriented toward print practices" [43]. While print is certainly not dead, for many, if not most, purposes it has been replaced by dynamic text on screens of varying sizes. To a large extent, the idea of the final printed document has thus become obsolete, and with it many related concepts such as pages and static page layout.

When the page-oriented document structure disappears, what remains? The structure of the *text*, i.e., the minimal structure necessary to dynamically display the text on a wide variety of displays and to interact with it.

For many applications, this minimal structure closely corresponds to that offered by Markdown and similar lightweight markup languages. Offering only Markdown for editing clearly reduces the number of functions an editor has to include and this significantly reduces the number of buttons and options a writer can press or choose from. This reduction visibly contributes to the reduction of toolbars: modern editors offer more space to actually write than those from the late 1990s.

For a long time it was believed that SGML and later XML would finally be used directly by end users when WYSIWYG editors would eventually become available. [34] But it turns out that the obsolescence of the printed document as the end goal of document preparation also rendered WYSIWIG obsolete. New tools explicitly separate writing and rendering, and the rendering is delegated to the machine. In fact, they are even *advertised* as not being WYSIWYG.

Andrew Tanenbaum famously complained: "WYSIWYG is a step backwards. Human labor is used to do that which the computer can do better."[5] In this sense, the development back to earlier practices can be seen as a step forward: writers are in control of their texts, but the formatting for presentation is taken over by the machine. Thus writers no longer feel the pressure to always maintain a neat, quasi-final document while writing.

This is in line with a development that we have observed for some years now: new writing and editing applications are advertised with a very strong emphasis on helping writers to "focus on the text," to offer "a unique writing experience that lets you concentrate and clarify your message," with an interface that "is crafted to cut out noise" (iAWriter[6]). Everything else that seemed so important only 15 years earlier is left to later, mostly automatic stages in the production process.

We also see experimental applications using recent technological possibilities to finally approach writing in a way earlier experiments tried and proposed but did not succeed: One such example is Tilio[7], which tried to implement ideas proposed by [39]. The project was cut short by the COVID-19 situation in 2020, but the technical feasibility has been shown in an alpha version, so we might see another attempt later.

Similarly, the integration of various features and services in support of *writing* rather than formatting (e.g., outlining, reference management, note-taking, data plotting, and synchronous and asynchronous collaboration and messaging) into a

[5] http://www.few.vu.nl/~ast/home/faq.html
[6] https://ia.net/writer
[7] https://tilio.app

single application (such as Scrivener[8], Author[9], or Zettlr[10]) can be seen as a comeback of some of the ideas of [15]. Liberated from the dictate of the printed document as final form, in which the microstructure of the text disappears behind the macrostructure of the document and its organization into pages, it now takes the center stage and demonstrates the power of the computer as a writing tool that goes beyond the mimicking of its historical predecessors.

6. Acknowledgements

This work has been supported by ZHAW DIZH Fellowship Call 2019. Parts of this paper were written during a stay at Studio Cascina Garbald of the Fondazione Garbald.

References

[1] Antenna House, Inc. 2019. *Introduction to CSS for Paged Media*. Antenna House. https://www.antennahouse.com/css.

[2] Atzenbeck, Claus. 2021. "Interview with Beat Signer." *SIGWEB Newsletter*, no. Winter (February). https://doi.org/10.1145/3447879.3447881.

[3] Bergin, Thomas J. 2006a. "The Origins of Word Processing Software for Personal Computers: 1976–1985." *IEEE Annals of the History of Computing* 28 (4): 32–47. https://doi.org/10.1109/mahc.2006.76.

[4] ———. 2006b. "The Proliferation and Consolidation of Word Processing Software: 1985–1995." *IEEE Annals of the History of Computing* 28 (4): 48–63. https://doi.org/10.1109/mahc.2006.77.

[5] Callender, E. David. 1982. "An Evaluation of the AUGMENT System." In *SIGDOC '82: Proceedings of the 1st Annual International Conference on Systems Documentation*, 29–35. New York, NY, USA: ACM Press. https://doi.org/10.1145/800065.801306.

[6] Calonne, David S. 2006. "Creative Writers and Revision." In *Revision: History, Theory, and Practice*, edited by Alice Horning and Anne Becker, 142–76. Reference Guides to Rhetoric and Composition. West Lafayette, IN, USA: Parlor Press.

[7] Chang, Emily. 2005. "eHub Interviews Writely." Weblog eHub. https://web.archive.org/web/20110722190058/http://emilychang.com/ehub/app/ehub-interviews-writely/.

[8] https://www.literatureandlatte.com/scrivener/
[9] https://www.augmentedtext.info/author
[10] https://www.zettlr.com/

[8] Daiute, Colette A. 1983. "The Computer as Stylus and Audience." *College Composition and Communication* 34 (2): 134–45. https://doi.org/10.2307/357400.

[9] Dale, Robert. 1989. "Computer-based Editorial Aids." In *Recent Developments and Applications of Natural Language Processing*, edited by Jeremy Peckham, 8–22. London: Kogan Page.

[10] ———. 1997. "Computer Assistance in Text Creation and Editing." In *Survey of the State of the Art in Human Language Technology*, edited by Giovanni B. Varile, Antonio Zamponelli, Ronald Cole, Joseph Mariani, Hans Uszkoreit, Annie Zaenen, and Victor Zue, 235–37. Studies in Natural Language Processing. Cambridge, New York, Melbourne: Cambridge University Press. https://doi.org/10.5555/278696.278806.

[11] Dale, Robert, and Shona Douglas. 1996. "Two Investigations into Intelligent Text Processing." In *The New Writing Environment: Writers at Work in a World of Technology*, edited by Mike Sharples and Thea van der Geest, 123–45. Berlin, Heidelberg, New York: Springer.

[12] Dam, Andries van, and David E. Rice. 1971. "On-line Text Editing: A Survey." *ACM Computing Surveys* 3 (3): 93–114. https://doi.org/10.1145/356589.356591.

[13] Dowling, Carolyn. 1994. "Word processing and the ongoing difficulty of writing." *Computers and Composition* 11 (3): 227–35. https://doi.org/10.1016/8755-4615(94)90015-9.

[14] Eisenberg, Daniel. 1992. "History of Word Processing." *Encyclopedia of Library and Information Science*, no. 49: 268–78.

[15] Engelbart, Douglas C. 1962. "Augmenting Human Intellect: A Conceptual Framework." Stanford Research Institute. http://sloan.stanford.edu/mousesite/EngelbartPapers/B5_F18_ConceptFrameworkInd.html.

[16] Fraser, Neil. 2009. "Differential Synchronization." In *Proceedings of the 9th ACM Symposium on Document Engineering*, 13–20. DocEng '09. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1600193.1600198.

[17] Geisler, Cheryl, Charles Bazerman, Stephen Doheny-Farina, Laura Gurak, Christina Haas, Johndan Johnson-Eilola, David S. Kaufer, et al. 2001. "IText: Future Directions for Research on the Relationship between Information Technology and Writing." *Journal of Business and Technical Communication* 15 (3): 269–308. http://jbt.sagepub.com/cgi/content/abstract/15/3/269.

[18] Gibson, James J. 1977. "The Theory of Affordances." In *Perceiving, Acting, and Knowing: Toward an Ecological Psychology*, edited by Robert Shaw and John

Bransford, 67–82. Hillsdale, NJ: Lawrence Earlbaum. http://www.worldcat.org/isbn/9781138203860.

[19] Haas, Christina, and John R. Hayes. 1986. "What Did i Just Say? Reading Problems in Writing with the Machine." *Research in the Teaching of English* 20 (1): 22–35. http://www.jstor.org/stable/40171057.

[20] Haigh, Thomas. 2006. "Remembering the Office of the Future: The Origins of Word Processing and Office Automation." *IEEE Annals of the History of Computing* 28 (4): 6–31. https://doi.org/10.1109/mahc.2006.70.

[21] Hawisher, Gail E. 1988. "Research update: Writing and word processing." *Computers and Composition* 5 (2): 7–27. https://doi.org/10.1016/8755-4615(88)80002-1.

[22] Hill, Charles A., David L. Wallace, and Christina Haas. 1991. "Revising online: Computer technologies and the revising process." *Computers and Composition* 9 (1): 83–109. https://doi.org/10.1016/8755-4615(91)80040-k.

[23] Holt, Patrik O'Brian, and Noel Williams, eds. 1992. *Computers and Writing: State of the Art*. 1st ed. Hardcover; Springer. http://www.worldcat.org/isbn/0792318587.

[24] Jokinen, Jussi, Aditya Acharya, Mohammad Uzair, Xinhui Jiang, and Antti Oulasvirta. 2021. "Touchscreen Typing as Optimal Supervisory Control." In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3411764.3445483.

[25] Kempen, Gerard, Gert Anbeek, Peter Desain, Leo Konst, and Koenraad De Smedt. 1986. "Author Environments: Fifth Generation Text Processors." In *ESPRIT'86 Results and Achievements*, edited by The Commission of the European Communities: Directorate General XIII: Telecommunications, Information, Industries & Innovation, 365–72. Amsterdam, New York, Oxford, Tokio: North-Holland.

[26] Kempen, Gerard, and Theo Vosse. 1992. "A Language-Sensitive Text Editor for Dutch." In *Computers and Writing: State of the Art*, edited by Patrik O'Brian Holt and Noel Williams, 68–77. Boston, Dordrecht, London: Kluwer.

[27] Kernighan, Brian W., and Michael E. Lesk. (1978) 1982. "UNIX Document Preparation." In *Document Preparation Systems*, edited by Jürg Nievergelt, Giovanni Coray, Jean-Daniel Nicoud, and Alan C. Shaw, 1–20. Amsterdam: North-Holland.

[28] Kitromili, Sofia, James Jordan, and David E. Millard. 2020. "What Authors Think about Hypertext Authoring." In *Proceedings of the 31st ACM Conference*

on Hypertext and Social Media, 9–16. HT '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3372923.3404798.

[29] Lai, Jianwei, Dongsong Zhang, Sen Wang, Isil Doga Yakut Kilic, and Lina Zhou. 2019. "ThumbStroke: A Virtual Keyboard in Support of Sight-Free and One-Handed Text Entry on Touchscreen Mobile Devices." *ACM Trans. Manage. Inf. Syst.* 10 (3). https://doi.org/10.1145/3343858.

[30] Mahlow, Cerstin, and Robert Dale. 2014. "Production Media: Writing as Using Tools in Media Convergent Environments." In *Handbook of Writing and Text Production*, edited by Eva-Maria Jakobs and Daniel Perrin, 10:209–30. Handbooks of Applied Linguistics. Berlin, Germany: De Gruyter Mouton.

[31] Mahlow, Cerstin, and Michael Piotrowski. 2009. "LingURed: Language-Aware Editing Functions Based on NLP Resources." In *Proceedings of the International Multiconference on Computer Science and Information Technology*, 4:243–50. Mragowo, Poland: Polish Information Processing Society. http://www.proceedings2009.imcsit.org/pliks/101.pdf.

[32] Meyrowitz, Norman, and Andries van Dam. 1982. "Interactive Editing Systems: Part I." *ACM Computing Surveys* 14 (3): 321–52. https://doi.org/10.1145/356887.356889.

[33] Piolat, Annie. 1991. "Effects of word processing on text revision." *Language and Education* 5 (4): 255–72. http://cogprints.org/3621/.

[34] Piotrowski, Michael. 2019. "History and the Future of Markup." In *Proceedings of XML Prague 2019*, edited by Jirka Kosek, 323–33. Prague. http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf#page=335.

[35] Ross, Donald. 1991. "Prospects for Writer's Workstations in the Coming Decade." In *Evolving Perspectives on Computers and Composition Studies*, edited by Gail E. Hawisher and Cynthia L. Selfe, 84–110. Urbana, IL, USA: National Council of Teachers of English.

[36] Rosson, Mary B. 1983. "Patterns of experience in text editing." In *CHI '83: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 171–75. New York, NY, USA: ACM. https://doi.org/10.1145/800045.801604.

[37] Severinson Eklundh, Kerstin. 1992. "Problems in Achieving a Global Perspective of the Text in Computer-based Writing." In *Computers and Writing: Issues and Implementation*, edited by Mike Sharples, 73–84. Boston, Dordrecht, London: Kluwer. http://www.jstor.org/stable/23370612.

[38] Sharples, Mike. 1996. "An Account of Writing as Creative Design." In *The Science of Writing. Theories, Methods, Individual Differences, and Applications*,

edited by C. Michael Levy and Sarah Ransdell, 127–48. Hillsdale, NJ, USA: Lawrence Erlbaum.

[39] ———. 1999. *How We Write: Writing as Creative Design*. New York, NY, USA: Paperback; Routledge. http://www.worldcat.org/isbn/0415185874.

[40] Sharples, Mike, James Goodlet, and Lyn Pemberton. 1989. "Developing a Writer's Assistant." In *Computers and Writing: Models and Tools*, edited by Noel Williams and Patrik O'Brian Holt, 22–37. Oxford: Intellect.

[41] Sharples, Mike, and Lyn Pemberton. 1990. "Starting from the Writer: Guidelines for the Design of User-Centred Document Processors." *Computer Assisted Language Learning* 2 (1): 37–57.

[42] Taylor, Lee R. 1987. "Software Views: A Fistful of Word-Processing Programs." *Computers and Composition* 5 (1): 79–90. http://computersandcomposition.osu.edu/archives/v5/5_1_html/5_1_8_Taylor.html.

[43] Van Ittersum, Derek. 2008. "Computing Attachments: Engelbart's Controversial Writing Technology." *Computers and Composition* 25 (2): 143–64. https://doi.org/10.1016/j.compcom.2007.12.001.

[44] Vernon, Alex. 2000. "Computerized Grammar Checkers 2000: Capabilities, Limitations, and Pedagogical Possibilities." *Computers and Composition* 17 (3): 329–49. https://doi.org/10.1016/s8755-4615(00)00038-4.

[45] Whiteside, John, Norman Archer, Dennis Wixon, and Michael Good. 1982. "How do people really use text editors?" *ACM SIGOA Newsletter* 3 (1-2): 29–40. https://doi.org/10.1145/966873.806474.

[46] Williams, Noel. 1990. "Writers' Problems and Computer Solutions." *Computer Assisted Language Learning* 2 (1): 5–25.

[47] Williams, Noel. 1992. "New Technologies. New Writing. New Problems?" In *Computers and Writing: State of the Art*, edited by Patrik O'Brian Holt and Noel Williams, 1–19. Boston, Dordrecht, London: Kluwer.

[48] Williams, Noel, and Patrik O'Brian Holt, eds. 1989. *Computers And Writing: Models And Tools*. Oxford: Hardcover; Intellect. http://www.worldcat.org/isbn/187151603X.

[49] Wohl, Amy D. 2006. "How We Process Words: The Marketing of WP Software." *IEEE Annals of the History of Computing* 28 (4): 88–91. https://doi.org/10.1109/mahc.2006.66.

Jiří Kosek (ed.)

XML Prague 2022
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 2022

ISBN 978-80-907787-0-2 (pdf)
ISBN 978-80-907787-1-9 (ePub)