A JavaScript Framework for Visual and Native XML Editors

Jochen Graf

Thesis submitted to
University of Cologne
Faculty of Arts and Humanities
for obtaining the degree
MAGISTER ARTIUM
in
Humanities Computer Science
(Historisch-Kulturwissenschaftliche Informationsverarbeitung)

Prof. Dr. Manfred Thaller

Table of Contents

Preliminary Remarks:

This thesis uses two forms of citations: in-text references and footnotes. In-text references are used to reference the list of works cited, to be found at the end of this thesis. The abbreviation [OCon10], for example, references a paper that is written by Martin F. O'Connor and is published in 2010. Secondly, footnotes with simple URLs reference to web tools that are of high importance but have no printed documentation available. These URLs may help the reader in easily finding the tool on the web, without guarantee for the persistence of the URL, though. All URLs mentioned in footnotes have been accessed on 13th December, 2013. The thesis often speaks about W3C standards or about other public web standards such as *XPath*, *XForms*, *TEI* or *XML*. Since these web standards are omnipresent on the web, it is sufficient to label them as such in the text without an explicit reference to their public URL. The URL of the W3C standard *Extensible Markup Language (XML) 1.0*[1] and the URL of the *Text Encoding Initiative (TEI)*[2] shall be representatively mentioned here. Proper names appear in great quantities, e.g. *XMLHttpRequest*. To keep the typeface sane, a proper name is italicised only if it contributes to a better understanding. The source code of the software described in this thesis is available on a public software repository[3].

1 http://www.w3.org/TR/REC-xml/
2 http://www.tei-c.org/
3 https://github.com/xrxplusplus

1. Introduction

The software industry is currently in the middle of a paradigm shift. Applications are increasingly written for the World Wide Web [Mikk07]. Once being mainly a platform for information access, the World Wide Web has turned into an application platform. The technology change, on the one hand, is driven forward by the browser specifications. With markup languages such as HTML5 or Scalable Vector Graphics (SVG), an increasing number of built-in browser technologies have become available for the development of rich internet applications [Toff11]. On the other hand, many efforts have been spent to optimize the JavaScript environment. JavaScript meanwhile has become a viable platform for the development of complex interactive web applications [Kuus09].

One community benefiting from the further development of the JavaScript environment is the XML community. Browser vendors never had and still have no larger interest in supporting the XML community, since the XML community represents a minority on the web [Kay11]. Although an XPath API is recommended for web browsers since 2004 according to the *DOM Level 3* W3C specification, there are still modern browsers that do not support an XPath API yet. The W3C specification XForms 1.0 (2003), an XML-oriented forms framework intended to replace HTML forms, has also not been implemented in most browsers to this day. With the advent of new web devices, e.g. mobile devices, e-book readers or Smart TVs, the web community and the XML community are drifting apart once again. As a consequence, client-side XML application development is still not as natural as in Java or C programming. XML processing is mainly seen as a server-side task in web programming.

In the years 2011 to 2013, a shift in direction is initiated. Within these years, the first stable JavaScript-based XML processors appear on the web. Google publishes its JavaScript XPath 1.0 implementation *Wicked Good XPath* in September 2012 [Den12]. Saxonica releases a client edition of its originally Java based XSLT 2.0 processor in June 2012 [Kay11]. Furthermore, a JavaScript based XQuery processor is provided by an academic project called *XQuery in the Browser* (XQIB), which is, like the Saxon XSLT processor, cross-compiled from Java into JavaScript [Ett11]. Finally, a first purely client-side XForms processor written in JavaScript appears in 2011, *XSLTForms* [Cout11].

Besides the browser becoming a viable environment for (XML) application development, there is another paradigm shift in progress, known under the buzzword *Semantic Web*. An increasing number of Semantic Content Authoring (SCA) tools have appeared within the last years [Khal13a]. The aim is to develop user-friendly editing tools that make the creation of semantically structured web pages easier for users that have no in-depth knowledge about semantic technologies or about markup technologies, so-called WYSIWYM[4] editors [Khal13b]. Technically, those tools rely on a

[4] WYSIWYM is an acronym for What-You-See-Is-What-You-Mean

browser built-in feature, *contentEditable*, also known as *HTML WYSIWYG*. The development of WYSIWYG or WYSIWYM editors also plays an important role within the XML community [Bin13]. Unlike in the Semantic Web community, the usage of web based tools is not as natural as in the Semantic Web community: the majority of user-friendly XML content creation is still done with desktop style XML editors [Flyn13]. Web-based WYSIWYG XML editors are rare at the moment and, due to the only incomplete XML support of browsers, subject of investigation. Especially the W3C recommendation XForms is currently seen as a viable solution to develop browser-based XML editor applications [Cam13].

Although browser-based XML processing is an innovative technology indeed, the JavaScript-based XML processors and tools mentioned above can be called outdated already at their time of publication, if one plays the devil's advocate. From a mature XML developer's point of view, they all rely on a former XML model, the Document Object Model (DOM), which forms the basis for all browser-based XML processors mentioned above. It is the only built-in XML data model available in a browser. Modern Java or C based XPath, XSLT and XQuery implementations, however, have not relied on the Document Object Model for a long time, but use streaming or binary XML APIs [Kay10] [Zhan06a]. Besides such modern implementations, with the W3C specification *XQuery 1.0 and XPath 2.0 Data Model* (XDM), there is also an abstract data model definition for the XML language available since 2010 that can replace the Document Object Model and takes streaming and binary XML processing into account.

The JavaScript framework for visual and native XML editors described here realizes the software-technical experiment of a purely client-side XML processing system that allows the development of native and visual WYSIWYM XML editors in the browser. The JavaScript framework behaves emulative in two respects: it firstly goes around the browser built-in DOM and realizes an XML processing system based on a streaming and binary XML representation model. Secondly, it goes around the browser built-in WYSIWYG feature and implements an own WYSIWYG control written in JavaScript. The software-technical experiment is guided by two fundamental questions: (1) Is the JavaScript environment powerful enough to make the development of visual and native XML editors independent from built-in browser features, i.e. from the further development of browser specifications and web devices? (2) Which basic software components are missing that would make the development of web-based visual and native XML editors as natural as in the Java or C world?

The work is organized as follows: Chapter 2 (*Related Work*) introduces the tools and also the discourse related to the subject visual and native in-browser XML editing. Three crucial concepts demanding clarification appear in chapter 2, MVC[5], WYSIWYG[6] and WYSIWYM, which are further specified in chapter 3 (*Terminology*). Chapter 4 (*Contributions*) and chapter 5 (*Definition*) sum up the preceding chapters. Here, the essential characteristics of the JavaScript framework for native and visual XML editors

[5] MVC is an acronym for Model-View-Controller
[6] WYSIWYG is an acronym for What-You-See-Is-What-You-Get

are listed to emphasize the similarities and also the dissimilarities to the related works described in chapter 2. With chapter 6 (*The Use Case*), the theoretical preliminaries are completed and the first main chapter starts. In chapter 6, it is argumented that a JavaScript framework for native and visual XML editors creates a special use case of an XML application in being a system that is able to synchronize different representations of one and the same XML instance, i.e. a *visual* representation with an according *native* representation and vice versa. At the end of chapter 6, it will be carved out that a specific XML representation model different from the browser DOM is the preferred solution for incremental XML synchronization, namely a *dynamic XML labeling scheme*. The XML labeling scheme is prepared in chapter 7 (*Warm-up: APIs for XML Processing*) and is described in more detail in chapter 8 (*A Token-Based Dynamic XML Labeling Scheme*). Chapter 9 (*XML Processing in JavaScript*) informs about implementation details of the XML editor framework, which realizes the use case described in chapter 6 and makes use of the labeling scheme defined in chapter 8. It also gives an overview of which parts of the use case could be realized already and which are subject of future work. Chapter 9 (*Conclusion*) summarizes the results.

2. Related Work

In the following section, I will discuss three tools that are related to the topic native and visual in-browser XML editing. Each tool introduces a specific technical or conceptual aspect. *Semantic Content Authoring Tools* (chapter 2.1) are higher level tools that deal with the collection of semantically structured web contents from ordinary web users. Ordinary web users in this context means technical non-expert users that have no in-depth knowledge about markup languages or semantic technologies. The second and third tool are lower level tools that mainly raise technical questions: To which extent do browser-based tools support XML at all? The W3C recommendation XForms plays an interesting role in this respect (chapter 2.2). How is it possible to visually edit text contents with today's browsers at all? Online code editors found an interesting technical approach for this (chapter 2.3). To compare the three tools, the following system is used:

- The original intention of the tool is described.
- The achievements regarding the main subject, native and visual in-browser XML editing, are summarized.
- The data model and the software components involved are described.
- The natural strengths and limitations of the tool are outlined.
- Insights about the technical challenges currently under development are given.
- Domain specific applications that use or adapt the tool for their special needs are introduced. In this context, the relation of the tool to the other two tools is also explained.

2.1 Semantic Content Authoring Tools

The starting point of Semantic Content Authoring tools is the assumption that the majority of information in the World Wide Web is still contained in unstructured documents. This 'Semantic Gap' is considered to be insufficient, whereas a ‘Semantic Web’ would have the following advantages [Khal13a]:

- Semantic content leads to better search and retrieval possibilities by means of faceted search and question answering techniques.
- Semantic content leads to more sophisticated information presentation and information integration systems, e.g. with the help of semantic overlays or semantic mashups.
- In a semantic web, personalized semantic portals as well as reusability and exchange of web data would become easier.

The motivation to develop SCA tools is based on the following arguments: (a) Text, images and videos are the natural way how humans interact with information in the web and we do not expect that this will change. (b) Gathering structured content is time-consuming. It is not easy to motivate users to do this additional work. (c) The manual and semi-automatic creation of rich semantic content is a less developed aspect of the semantic content life-cycle. Thus, (d) the acceptance and dissemination of Semantic Web technologies depends on the availability of easy to use authoring tools. [Khal13a]

Having the issue native and visual in-browser XML editing in mind, the main achievement is the description of a conceptual framework including quality attributes for SCA tools. The framework includes (a) visualization techniques such as highlighting and associating of semantic data; (b) visualization binding techniques (progressive, hierarchical or grouping visualizations) that aim to use similar visualization techniques for similar semantics; (c) authoring techniques such as form editing, inline editing or drawing; (d) helper components for automation, recommendation or real-time collaboration that cannot be implemented purely client-side but need a server-side component [Khal13a] [Khal13b].

Example 1: Semantically Enriched HTML Document Using *Schema.org*

```
<div itemscope itemtype="http://schema.org/Offer">
  <span itemprop="name">Blend-O-Matic</span>
  <span itemprop="price">$19.95</span>
  <img src="four-stars.jpg" />
  Based on 25 user ratings
</div>
```

The basic data format used by SCA tools is HTML. Semantic markup is integrated with the help of attribute level extensions. Older versions like Microformats or embedded RDF re-purpose existing markup definitions, particularly the HTML class attribute, and in that follow a non-standard annotation strategy. This approach is considered limited, since data constructs cannot be validated in the absence of proper grammars used for their definition [Khal13a]. Newer formats such as RDFa and HTML5 Microdata introduce standard annotation strategies, i.e. a fixed set of attributes complemented by Metadata standards such as *Dublin Core* or *Schema.org* (example 1).

Technically, SCA tools firstly rely on a browser built-in component, namely *contentEditable*, for some browsers also called *designMode*, which allows the edition of HTML documents by means of mouse and keyboard interactions, that is by a cursor caret. Secondly, they make use of JavaScript libraries such as *CKEditor*[7] that add convenience APIs and plug-in systems on top of *contentEditable* [Hees10]. These JavaScript libraries offer menus and icon bars that make the edition of HTML documents as user-friendly as with a rich text editor. The software technical challenge of a SCA tool is to implement a single-entry-point user interface for the edition and exploration of semantically structured documents and to complement this interface with server-side helper components. An overview and evaluation of current SCA tools is given in [Khal13a].

The strength of SCA tools results from their usage of the *contentEditable* feature. SCA tools benefit from the circumstance that *contentEditable* has been available in all browsers for a long time. In reusing a browser built-in feature and existing JavaScript UI libraries, the costs for implementing SCA tools are relatively low.

Example 2: *New Testament Virtual Manuscript Room* Editor

```
<span                    wce="__t=unclear&amp;__n=&amp;unclear_text_reason=poor
%20image&amp;unclear_text_reason_other=&amp;insert=Insert&amp;cancel=Cancel"
class="unclear" wce_orig="text">text</span>
```

Semantic Content Authoring tools are influential on the XML community. Tools that intend to gather structured text documents from ordinary web users according to the XML standard *Text Encoding Initiative* (TEI), for example, follow the SCA approach. Since *contentEditable* does not support the editing of an XML document directly but only through HTML, TEI documents are transformed into an according HTML representation for edition beforehand. Example 2 shows an HTML snippet of the *New Testament Virtual Manuscript Room*[8] transcription editor. In the snippet, the HTML representation of a TEI element `<tei:unclear/>` is shown. The element name `unclear` is encoded in the HTML `class` attribute, whilst several attribute definitions of the original TEI `unclear` element are transcoded into a custom `wce` attribute with the help of a name-value-pair organized query string. For persistence and data exchange, the HTML construct is transcoded back into an XML/TEI document. The advantage of the SCA approach from the XML communities' point of view is the fact that all XML tags are

[7] http://ckeditor.com/

[8] http://ntvmr.uni-muenster.de/transcribing

hidden during authoring and users can edit structured texts as conveniently as with a rich text editor.

While the SCA approach and the browser built-in HTML WYSIWYG feature are widely used in the XML community, online code editors (see chapter 2.3), for example, turn away from *contentEditable* for technical reasons. Online code editors explicitly need cross-browser support for cut and paste from external applications and at the same time depend on a clean and consistent HTML construct, which is a known problem for *contentEditable* [Hav07].

2.2 XForms

XForms is a W3C recommendation intended to be the successor of HTML forms. It follows a model-view-controller design in separating the application and the data logic portion of a form-based web application from the view. With the establishment of a separate model, XForms is able to overcome several limitations of HTML forms [Dub03]. For example, HTML form controls are limited in respect to typing information. The values of form controls are just string values which have to be converted to the types expected by the application logic on the server. Missing type information also implies lack of client-side validation mechanisms [Fons07]. Thus, data validation as well as many other dynamic aspects of HTML forms can only be realized with the help of an additional scripting language, i.e. with JavaScript. With XForms, the amount of JavaScript needed to develop data-intensive browser applications can be reduced by means of a model-view-controller system and an XML user interface description language that together cover common patterns in data-centric interactive UI design.

The XForms recommendation was never accepted by the browser vendors, though. Most browsers do not support XForms natively up to this day. In recent years, however, XForms has been getting attention again by taking on a key role in the so-called XRX architecture, an acronym for XForms, REST and XQuery. See [McCr08] [Lain12] [Nem12] [Cam13], to mention but a few. XRX is a web architecture that uses XML and the XML technology stack from front to back, for data as well as for application logic. As such, XRX, on the one hand, fits well with the NoSQL movement, since it uses the same data model on the client and on the server. Through this, the overhead of a middleware for data mapping is unnecessary, and client-server communication is possible via simple REST interfaces. In using less technologies, less data models and less programming paradigms, instead relying on only one model—the XML technology stack—XRX also picks up general considerations [Kuus09] [Hev09] and criticisms [Mikk07] about up to date web application development, which can be summarized under the term ‘unified web application development’ [Lain11]. In the XRX architecture, XForms is consequently not only seen as a framework for form design, but plays the role of a browser-based editor framework for XML data following a declarative programming style. It can handle live-XML-instances in the browser and their original structure and contents can be dynamically modified through user interactions [Maal13].

In respect to the subject native and visual in-browser XML editing, XForms mainly contributes to the aspect of *native* XML support. XForms is a useful technology available on the web that allows native in-browser XML editing to a broader extent, including XPath and partial XML Schema support. Native in this context means that the original XML document requested from the server needs not to be transformed into an HTML representation before editing can take place, as is the case for SCA tools, but can be modified directly. Since XForms integrates a set of form controls to describe the visual part of XML editing applications, XForms also serves as a user-friendly *visual* XML editing tool. The XML markup is always hidden from the user.

Example 3: A Basic XForms Document

```
<html>
<head>
  <xf:model schema="./data.xsd">
    <xf:instance id="i1">
      <data><e>3</e></data>
    <xf:instance>
    <xf:instance id="i2">
      <data><e></e></data>
    </xf:instance>
    <xf:submission action="./save.php" ref="instance('i1')"/>
    <xf:bind id="b1" ref="instance('i1')//e"/>
    <xf:bind id="b1" ref="instance('i2')//e"/>
    <xf:bind nodeset="//e" constraint="xs:integer(.) gt 2"/>
  </xf:model>
</head>
<body>
  <xf:input bind="b1"/>
  <xf:textarea bind="b2"/>
</body>
</html>
```

Example 3 shows a simplified XForms document. The XForms Model `<xf:model/>` contains all XML data, held by instance elements (`<xf:instance/>`), and the application logic of a form. The data types of XML instances can be defined in an XML Schema document linked in the `<xf:model/>` element. The application logic portion includes data submission definitions (`<xf:submission/>`) and Model Item Properties (MIP) written in XPath (`constraint="xs:integer(.) gt 2"`). With MIPs, dynamic calculations and constraints on instance data can be defined that can not be expressed with the XML Schema language. The XForms view, located in the HTML `<body/>` element, defines a standard control set to declaratively describe the visual part of a form. Form controls are bound (`<xf:bind/>`) to instance data by means of XPath expressions. A detailed XForms introduction is given by [Dub03].

Since there is no acceptance by the browser vendors, XForms implementations appear in the form of browser plug-ins (X-Smiles [Honk07]), server-side processors (Orbeon Forms[9], betterFORM[10]) and recently as purely client-side processors written in JavaScript. The first stable JavaScript implementation, XSLTForms [Cout11], appeared in 2011. XSLTForms includes an XPath 1.0 processor written in JavaScript. XML data manipulation is done by means of the HTML DOM update functions. Server-side implementations use standard Java XML APIs for all kinds of XML processing such as querying, updating and validating.

The natural strength of XForms is its usefulness for data-intensive web applications [Pohj10]. XForms supports arbitrary XML documents and is—unlike the SCA tools described above—independent from an application-specific data model[11]. A limitation of XForms from the perspective of an XRX developer is that it can indeed handle XML data, but not XML documents in a closer sense. XForms first of all allows to *display* data structures and to modify and validate data *values*, but only for a structurally fixed XML encoded form. XForms indeed has some basic support for the edition of structures by means of the `xs:insert` and `xs:delete` actions, which are mainly used in combination with repeating data sets. The edition and validation of semi-structured XML fragments, i.e. the edition of so-called XML Mixed Content, is not supported natively by XForms, however [Maal13].

Current investigations and software technical efforts are thus engaged with the implementation of XForms controls for the edition of XML Mixed Content as another generic UI component extending the XForms control set [Maal13]. Closely related with the authoring process of semi-structured data is the question of structural validation as well as Schema-driven tag suggestions, sometimes also called 'target markup adoption' [Flyn13]. XML Schema implementations do not provide standard interfaces to generate context sensitive tag suggestions. Hence, [Maal13] for example, suggests a 'try and tell' method: if one temporarily inserts an element or an attribute node into an XML document at a specific position, of which one certainly knows that it is invalid at this position, Schema processors normally report an error message but also a list of nodes allowed at this very position. This is a possible workaround to extract tag information contained in XML Schema documents, which at least works in combination with a specific XML Schema processor. A generalization of the try and tell method would depend on consistent error reports across XML Schema processors, which is not given so far.

Since the user interface and the XML Schema document grow more and more together within the XForms approach, some investigations experiment with the automatic generation of complete web user interfaces out of XML Schemas [Cam13]. Such investigations are rather concerned with the markup level, i.e. with the UI

[9] http://www.orbeon.com/
[10] http://betterform.de/
[11] Although SCA tools support different ontologies for semantic content authoring, their basic data model is always HTML.

description language of the XForms standard than with the internal XML processing model and are thus not further discussed here.

Figure 1a: Mixed Content Annotation with an XForms Tool

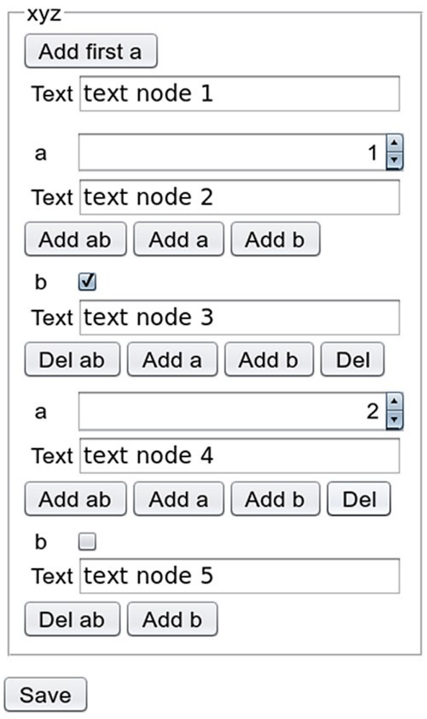
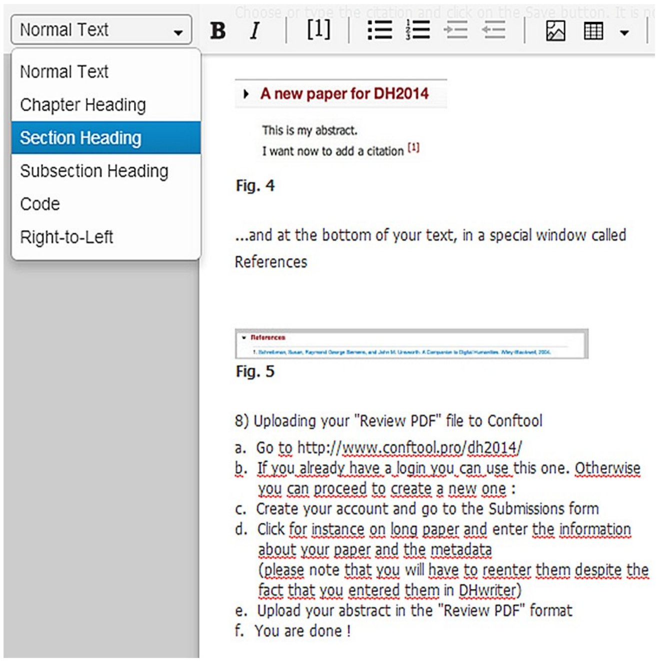


Figure 1b: Mixed Content Annotation with an SCA Tool



A new paper for DH2014

This is my abstract.
I want now to add a citation [1]

Fig. 4

...and at the bottom of your text, in a special window called References

Fig. 5

8) Uploading your "Review PDF" file to Conftool

a. Go to http://www.conftool.pro/dh2014/
b. If you already have a login you can use this one. Otherwise you can proceed to create a new one :
c. Create your account and go to the Submissions form
d. Click for instance on long paper and enter the information about your paper and the metadata (please note that you will have to reenter them despite the fact that you entered them in DHwriter)
e. Upload your abstract in the "Review PDF" format
f. You are done !

Figure 1a and figure 1b show user interface controls for the annotation of mixed XML content, respectively mixed HTML content. The first tool is based on an XForms processor [Maal13], the second is a TEI editor following the SCA approach[12]. Whilst the XForms tool is outstanding in utilizing XML Schema information for document validation and document enrichment by means of automatic tag suggestions, the SCA tool wins over with its user-friendly rich text interface.

Compared to the SCA tools described above, there are many aspects absent when using XForms as a tool for semantic content authoring. Especially advanced visualization techniques and drawing components are missing. XForms is designed to be an extendible framework, current XForms implementations do not yet offer a convenient way to add custom UI components, though. Nevertheless, some prototypical XForms-based transcription and annotation tools that support XML Mixed Content and to some degree semantic content authoring, e.g. the *teian* editor[13] or the *XRXEditor* [Ebn13], do exist.

[12] http://dhwriter.org
[13] http://teian.sourceforge.net/

2.3 Online Code Editors

Implementing a JavaScript framework for visual and native in-browser XML editors starts with the selection of a basic editing control. Browsers offer two built-in alternatives for this: HTML forms and *contentEditable*. Firstly, HTML forms still define the main interaction capabilities for web applications. Since form controls only offer a single font and style that cannot be changed with JavaScript, they are less attractive for applications that require advanced text visualization techniques, for example SCA tools. The second built-in alternative thus is to use *contentEditable*, which offers an interactive WYSIWYG editing control supporting all structural semantics and formattings of the HTML and the CSS language.

Modern Online Code Editors, apparently the first editing tools available on the web, offer a third alternative for visual in-browser text editing. The crux is that they do not just reuse a browser built-in editing feature, but emulate a visual editing control in many parts. In respect of their emulative nature, modern online code editors can be vaguely compared with cross-platform desktop UI frameworks like Qt or Java Swing. Code editors render a browser editing control themselves, just like desktop UI frameworks render user interface components that are normally provided by the operating system. The approach is slightly different, however, due to the specific conditions of the browser environment: (1) code editors do not render the user interface on the pixel level, as is the case for desktop UI frameworks, but the overall visual look and feel of a control is mainly realized by means of the HTML and CSS language. Some lesser components that cannot be expressed with the HTML and CSS language have to be drawn specially with JavaScript. For example, the cursor caret is realized with JavaScript in form of an ever blinking vertical line. (2) Browsers do not provide a standard programming interface for keyboard and mouse interactions, as is the case for operating systems. Browsers offer an event API just in combination with built-in browser controls, e.g. form controls, but not independent of them. This limitation can only be resolved by means of a so called ‘hidden textarea approach’. The hidden textarea approach makes the browser act as if there is an interactive control in the HTML page that is focusable, has support for copy and paste and can receive keyboard and mouse input [Hav12b], although the user is in reality interacting with an emulated editing field provided by the code editor application. Example tools that implement the hidden textarea approach are *CodeMirror*[14] and *Ace*[15].

The natural strength of these tools is to combine the stable cross-browser behavior of HTML forms with the rich look and feel of *contentEditable*. HTML form controls behave more stable within the authoring process due to the fact, for example, that content pasted from an external application is interpreted as plain text by a browser, whilst content pasted into a *contentEditable* field is interpreted as formatted text by default. Formatted text pasted via the browser’s clipboard into a *contentEditable* field is automatically transformed into an HTML plus CSS construct. But there exist no cross-

[14] http://codemirror.net/
[15] http://ace.c9.io/

browser standardizations of, how exactly the transformed HTML construct should look. Applications such as code editors that aim to produce plain text documents in the end and use HTML only as an interlayer for text highlighting, thus find favor with the hidden textarea approach that internally uses an HTML form for authoring [Hav07]. Since a visual and native XML editor also sees HTML just as an intermediate layer for visualization, the code editor approach is attractive for this use case as well.

Limitations for visual in-browser text editing result from the inherent line-based approach of code editors, which makes them only partially useful for rich text authoring. Each line of code is wrapped by one or more HTML DIV or SPAN tags in a code editing field. This cooperates well with single-line, word-based or character-based rich text elements such as headings, lists, paragraphs, line spacing or font styles, but less with multi-line components like tables or trees. Support for multi-line rich text authoring does not arise in a natural way as with *contentEditable* and is not yet implemented in the world of emulative browser editing controls so far.

Other challenges currently under development are real-time collaboration and, in combination with that, the rendering of multiple cursors and cursor selections within one control. A domain-specific tool that uses an online code editor as its basic editing field is the *brat rapid annotation tool* [Sten12]. Surprisingly, the annotation tool is not intended to be a text authoring tool, but a pure annotation tool with a fixed text that cannot be changed. Annotations can be inserted by means of keyboard and mouse interactions and are visualized with different text formattings and with inline SVG widgets inside the code editing field. The example shows that the JavaScript-based emulation approach of a browser-based rich text editing field brings out an interesting alternative for *contentEditable*. The approach is currently mainly implemented by code editors, but can be transferred to other contexts.

2.4 Summary

When investigating the landscape of current web tools for the authoring of semantically structured contents, one can identify two different approaches that evolve in different communities, widely independently of each other: the SCA approach in the Semantic Web community and the XForms approach in the XML community. Whilst the SCA approach has a natural focus on the user interface part of an editor application and naturally supports text annotation by means of easy accessible keyboard and mouse interactions, XForms is outstanding on the model side in supporting arbitrary XML formats and in utilizing XML Schema information within the authoring process for document validation and enrichment. There are web-based end-user applications such as transcription tools or text annotations tools that individually follow the one or the other approach. A project with explicit need for deeply nested and complex annotations would choose the XForms approach, whereas a mostly rich-text oriented project with flat semantics would better operate with an SCA tool. In the need to decide about one of the two approaches, it is denoted that current web-based XML editing tools are not mature applications yet, rather supporting only parts of the features needed for specific authoring projects. For XML Schema-oriented projects, browser-based XML editors especially compete with and rank behind existing desktop XML editors, which up to today represent the quasi standard for user-friendly XML content creation [Flyn13].

In the following, I will investigate why the development of a web-based native and visual XML editor is not as natural as in the Java or C programming world. The investigation is constructed as a criticism on two browser built-in features, the browser DOM and the HTML WYSIWYM feature *contentEditable*. A solution arises from the JavaScript language, which is able to emulate browser built-in features and thereby can overcome their limitations.

3. Terminology

When introducing the tools that are related to the subject visual and native in-browser editing in the previous chapter, three concepts appeared: Model-View-Controller *(MVC)* in conjunction with XForms, What-You-See-Is-What-You-Get *(WYSIWYG)* as an often used browser built-in editing feature and What-You-See-Is-What-You-Mean *(WYSIWYM)* along with Semantic Content Authoring tools. These concepts, on the one hand, contribute to a better understanding of the subject. On the other hand, they are vague terms, as they are used as buzzwords in many software related contexts. In the following I will try to clarify what these terms mean in the special context of a native and visual XML editor.

3.1 MVC

The term Model-View-Controller (MVC) is an often quoted term which combines a multitude of ideas in software development. Sometimes, MVC is called a pattern, sometimes a design principle and sometimes a software architecture combining a whole diversity of patterns [Burb87] [Steel04] [Fowl06] [Pham10]. A first clarification can be reached by distinguishing server-side MVC and desktop MVC.

Server-side MVC is a mechanism used to manage the data flow within a data-intensive client-server application. Assuming a classical three-tier web architecture, there are typically three copies of a data item involved: a copy representing the (1) *record state* of a data item. This copy resides in the database for persistence and can be shared by multiple clients. The middle-tier of a web application typically holds a second copy, which represents the (2) *session state*. It is a temporary server-side version of a data item, on which a client is actually working, and which can be used for validation or type conversion mechanisms. The third copy represents the (3) *screen state*. It lies in the GUI components themselves and can be manipulated by means of user interactions [Fowl06]. Such server-side model-view-controllers realize a data binding concept: any change within the screen state of a data item is immediately propagated to the session state one level deeper. Further, the session state is automatically aligned with the record state whenever the screen state changes. The strength of the server-side MVC concept is a well-defined workflow for data-intensive applications. It sources out most of the data flow complexities from application development into the framework.

However, the MVC principle originally is a feature introduced by the object-oriented Smalltalk-80 programming language in the late 80’s [Burb87]. MVC here is a design principle not for web applications but for desktop applications, useful to free views and interactive controls of data logic information and to make controls more independent from each other, which was a common design problem in software development at that time. The principle became well known under the buzzword ‘separating data and presentation’. The original Smalltalk MVC idea is closely related to the appearance of multi-windowed computer applications, which were a novelty then. For example, a software application for architects or a Geographical Information System may manage several domain objects containing the raw data about a building or an urban landscape. The software also may offer different views and editing controls for these raw data—for example maps, 3D reproductions, tables or charts—each view and control set in another window. With the help of the Smalltalk MVC principle, every change of the raw data or every change made in one of the currently opened interactive windows, automatically leads to an update of the domain object instances as well as to an update of all other views currently opened.

Looked at in this way, desktop MVC is, on the one hand, similar to server-side MVC, since different copies and representations of one and the same data-set exist in an application and those have to be synchronized automatically. On the other hand desktop MVC is different, since the data copies are not spread over different nodes of a network but over multiple windows of a single computer application following an object-oriented design. Also the motivation is different: while the separation and synchronization of different data instances arises out of the physically distributed nature of a client-server architecture, the original MVC idea is artificial in this respect. It assumes that self-contained domain objects modeling the real world exist. Independent from that, generic UI controls exist to present and modify these domain objects. Domain objects and generic controls can be conveniently combined to constantly new domain-specific applications. Desktop MVC systems contribute to better code modularity, since new views and controls can be added to an application without breaking the application as a whole. Like for server-side MVC, data complexities are handled by the MVC system and are not in the responsibility of application developers.

3.2 WYSIWYG

WYSIWYG editors, from the outset, are counted as typical use cases implementing a model-view-controller [Burb87], since they are systems that allow data modeling through an interactive view, which automatically and incrementally updates a data object in the background. Above that, the term WYSIWYG became popular for computer applications in the print industry to describe a system in which textual and graphical contents displayed on screen during editing appear in a form closely corresponding to its appearance when printed [Khal13b]. Within the Apple culture, wherein the design of intuitive and easy to use user interfaces plays an important role, the term WYSIWYG is also used in a broader sense: as a way of risk-free experimenting with and proofing a layout, e.g. by means of moving elements of a document around with drag-and-drop, and also as a claim to get immediate feedback for all user interactions [Tsai98].

Again there are similarities: the model-view-controller principle in the WYSIWYG context synchronizes not multiple, but exactly two representations of a data set: a human-readable visual representation that can be edited interactively, and a computer-readable document in the background containing the raw formatting and structural layout data. The latter can never be manipulated directly by the user, but only through the visual interlayer on the screen. There are also conceptual differences between the original MVC idea and the WYSIWYG idea, though. The assumption of a self-contained independent domain object that models the real world is not pronounced by the WYSIWYG concept. All in all, the model side of an application is not of special interest for WYSIWYG. There is also a difference on the view side: the interactive editable view does not consist of generic controls and multiple views like in the original MVC idea, but is normally one single and application specific control. The main feature for WYSIWYG applications seems clearly to be a feature-rich interactive view, whereas the model side is of low interest and left to the respective application.

3.3 WYSIWYM

Example 4: Data Model of a WYSIWYM Editor

```
<article lang="en-US" xml:base="">
  <heading></heading>
  <section view="normal">
    <heading></heading>
    <figure/>
    <paragraph>Some <strong>bold</strong> text ...</paragraph>
  </section>
  <section view="accordion">
  </section>
</article>
```

With the What-You-See-Is-What-You-Mean (WYSIWYM) concept appearing in the middle of the 90's, the original MVC idea and the WYSIWYG world started to converge. WYSIWYM extends WYSIWYG with two aspects: firstly, a more intensive focus on the model side of a WYSIWYG application is established by supporting application independent file formats such as XML and, through this, support for semantic content authoring is added.

Web-based WYSIWYM editors preserve the meaning of elements, e.g. page headers and paragraphs are named as such (example 4). Unlike normal HTML WYSIWYG editors, the produced XML file is not only useful for web presentation but also for PDF creation or data interchange. WYSIWYM breaks the single monolithic application paradigm. Instead, a multitude of specialized tools can be applied towards document presentation and creation [Tsai98]. Technically, this makes a separation of UI controls and data model items necessary and, at the same time, two binding concepts: (1) a data binding concept to synchronize the model and the view, that is, a model-view-controller, and (2) a visualization binding language such as CSS to define the visual appearance of specific data items [Bin13]. Once a separation of the model and the view and also a data and a visualization binding mechanism is technically established, semantic content authoring becomes possible, where data items can be combined with arbitrary controls and with arbitrary visualization techniques [Khal13b]. In a WYSIWYM application, the model side as well as the view side can be extended with new data elements and generic components independently from each other. Thus, WYSIWYM is very close to the original MVC idea described in [Burb87].

3.4 Summary

In the following, I will use specific interpretations of the terms MVC, WYSIWYG and WYSIWYM. In the context of a visual and native XML editor, it makes sense to use the terms in the following way:

- A native and visual XML editor is not related to the server-side MVC concept, since all data synchronization is done within a single desktop application—a web browser.
- Instead, a native and visual XML editor closely follows the original Smalltalk MVC idea, where self-contained, domain-specific data models play an important role, and data items can be bound to multiple views consisting of multiple editing controls. The model-view-controller is responsible for synchronizing all these data instances and data controls.
- The WYSIWYM concept is more significant for a native and visual XML editor than WYSIWYG since WYSIWYM, just like MVC, integrates the idea of arbitrary, application independent and self-contained data models—for example an XML standard. The model side is not represented enough in the view-centric WYSIWYG concept to be of substantial meaning.
- Although WYSIWYM editors are sometimes counted as typical use cases implementing a model-view-controller, mainly due to the incremental data synchronization feature, I will regard WYSIWYM and MVC explicitly as two different concepts: (a) WYSIWYM is the concept of synchronizing exactly one

single and special data editing control—a rich text-like control—with a single corresponding data item—an XML Mixed Content fragment; (b) a model-view-controller is the concept of synchronizing a whole set of data controls (maybe a set of WYSIWYM controls) organized in views with one or more data items organized in data instances.

- The WYSIWYG concept is significant for a visual and native XML editor in its broader sense as a way of risk-free authoring of XML documents and the claim to get immediate feedback for all user interactions. I will subsume these ideas under the MVC and WYSIWYM concept, however.

4. Contributions

In this chapter, I will shortly summarize what a JavaScript framework for visual and native XML editors contributes to the area of browser-based tools for visual and user-friendly authoring of structured (XML) documents. Also, some first criticisms about the browser DOM are outlined here, which will conclude the theoretical part of this work and will lead over to the technical part.

WYSIWYM: visual, risk-free XML authoring. The JavaScript framework will extend a browser in such a way, that XML instances containing so-called XML Mixed Content can be loaded into and manipulated with a browser application. At the same time, an HTML construct forming a visual interlayer between the user and the XML document is provided, which constitutes an interactively editable representation of an XML Mixed Content fragment. This layer is called a 'WYSIWYM control' from here on. The WYSIWYM control takes part in a model-view-controller system, so that the data of the interactive control and its corresponding XML Mixed Content fragment are synchronized automatically. The HTML construct is generic: it works for arbitrary XML Mixed Content fragments. In following the emulative approach of online code editors as described above, the WYSIWYM control builds an alternative to the browser built-in *contentEditable* feature. It overcomes some limitations of *contentEditable* and the HTML DOM described in more detail below. The WYSIWYM control allows risk-free and user-friendly authoring of XML Mixed Content: markup is generally hidden from the user and well-formedness of XML markup is guaranteed by the application. A variety of techniques to visualize XML data in a WYSIWYM control [Bin13] exist. Those techniques lie in the area of common JavaScript programming and are thus mentioned from time to time, but are not further investigated here.

MVC: Incremental synchronization of UI controls and XML instances. Domain-specific XML Schemas, like the Schema of the Text Encoding Initiative (TEI) for example, typically integrate a mix of data-centric (structured) as well as text-centric (semi-structured) parts. Data-centric document parts are typically made editable with the help of form-like user interfaces, whereas text-centric parts are bound to rich text-like controls. Thus, visual XML editors typically do not assume a single editing area as is the case for most rich text editors, but rather multiple controls in one view which are bound to different nodes of an XML document. The JavaScript framework for visual and native XML editors implements a set of classes representing a model-view-controller, which can be used to synchronize a set of editing controls organized in views with an arbitrary number of data items organized in XML instances. HTML and JavaScript components, e.g. HTML forms, jQuery or Dojo controls, which can make use of the MVC classes, automatically take part in the MVC data synchronization flow. The JavaScript framework explicitly does not implement user interface components from scratch. The crux is to offer the possibility of adding native XML support to existing browser UI components that do not support XML themselves.

XML Representation Model. A visual and native XML editor, on the one hand, is an ordinary XML application that follows standard XML APIs, for example XPath for data binding or XML update for authoring, but on the other hand,—in one aspect—is a very special XML application. The crux of a native and visual XML editor results from the WYSIWYM and MVC concept: The main task is to handle and synchronize different physical copies of one and the same logical XML instance, i.e. to synchronize a *visual* XML representation with an according *native* XML representation and vice versa. To realize such a synchronization feature, a concept of 'sameness of XML nodes' is needed in the end, which amongst others plays an important role in the context of XML versioning systems [Lind04]. How can one identify, for example, whether the visual representation of an XML node currently edited by a user is logically 'the same' as the corresponding, but physically distinct node of the raw XML document, both held in the browser? Or how can one identify that two remote clients, which are under way to author the same XML document, are currently editing 'one and the same' XML node?

The central contribution of the JavaScript framework for visual and native XML editors is thus its support for an XML representation model realizing a concept of sameness of XML nodes across XML versions that shall stay synchronous at any time. Without going too deep into the technical details of XML representation models here, it seems obvious that the Document Object Model alone is inadequate for this use case. DOM creates an in-memory representation of an XML document and allows node calculations within this in-memory representation, but not across. This is due to the fact that the DOM does not know a concept of public node identifiers, instead working with internal memory pointers that are hidden outwards. On the contrary, XML representation models based on labeling schemes serve publicly available node identifiers by nature, which can be used—under certain conditions to be described in more detail later—by XML applications to compute node sameness across XML versions.

Since the Document Object Model is rejected as the basic XML processing model and an alternative is proposed instead, the JavaScript framework also has to contribute with an XPath implementation as well as a set of update operations based on the developed XML representation model.

An XML Schema implementation in JavaScript would be of great interest as well, as described in the context of current XForms authoring tools, but unfortunately does not lie within the realms of possibility of this work. There is currently no browser-based Schema implementation available on the web. Also, the cross-compilation of an existing Java-based Schema processor would not be an adequate solution, since XML editors emerge with so-called partial and incremental XML validation mechanisms [Nic03]. It is an unacceptable situation if a whole XML document has to be revalidated any time a user changes a single node value or inserts a single new node into the document. Partial and immediate validation in combination with frequent XML updates is a task, however, that researchers are just beginning to look at [Gor11]. Current XML Schema processors are designed for full document validation only. In addition, an interface for Schema-driven node suggestions is missing in today's Schema implementations, as described above. All in all, a JavaScript-based XML Schema processor fulfilling the needs of a native and visual XML editor framework requires many extra investigations, which have to be described elsewhere for the reason of space.

5. Definition

Characteristics of a native XML editor:

- holds and manipulates XML documents directly in the browser. XML processing is done purely client-side with JavaScript
- does not require an editor-specific XML format, but supports arbitrary domain-specific XML standards
- can handle all kinds of XML documents and all kinds of XML nodes, especially so-called XML Mixed Content is supported
- does not apply any changes or transformations on XML documents for technical reasons. It leaves the XML document untouched until the user manipulates it
- communicates with server-side applications via a simple REST interface, no server-side transformation services are required
- relies on the XML technology stack: it uses XPath for data binding, XML Schema for document validation and target markup adoption, and the XForms standard as an ideal for a browser-based XML model-view-controller system
- in any case uses XML as data format but also offers an XML-based user interface description language for common patterns, e.g. to describe application logic (data binding, data submission) and views (select box, textarea, input field, WYSIWYM)

Characteristics of a visual XML editor:

- is different from a code editor: the possibilities for visualizing XML exceed XML syntax highlighting. Visual XML means to hide XML syntax and not to highlight XML syntax

- editing visual XML is risk-free, guarantee of well-formedness is handled by the application
- integrates elements of rich text editors, forms, rich internet applications as well as drawing components
- uses standard browser technologies such as HTML, CSS, JavaScript or SVG, no plug-ins such as Java Applet or Flash are necessary

6. The Use Case

In the previous chapters it was carved out that a visual and native XML editor is a special use case in being a system that has to deal with different copies of a single XML instance, which have to be synchronized incrementally. In the following I will describe three points of the use case in more detail. (a) The differences between an *XML* model-view-controller and an ordinary non-XML MVC system are described. (b) The functionality of the browser WYSIWYG feature *contentEditable* and its ability for an 'ideal' XML WYSIWYM control is investigated. (c) Some first considerations are made about the XML update language relevant for a native and visual XML editor compared to other 'ordinary' XML applications, and also an estimation for which document size incremental updates should work.

6.1 XML Model-View-Controller

The concrete design of a model-view-controller depends on the particular data model supported as well as the data binding language used. A model-view-controller for XML, which is a hierarchical and semi-structured data format in combination with XPath, originally designed to address parts (not single nodes!) of an XML document, as the data binding language, creates a special situation compared to an ordinary MVC system. To filter out the main characteristics of an XML model-view-controller, I will compare the XML MVC with a prototypical object-relational MVC, which might not exist in exactly this form, but serves well for clarification. For this section, the following XML snippet and an according table are used:

Example 5a

```
<data>
  <a>Value 1</a>
  <a c='on'>Value 2</a>
  <b>Value 3</b>
</data>
```

Example 5b

Table: 'data'	a	b
1	Value 1	Value 3
2	Value 2	

6.1.1 Uniqueness of Data Binding Expressions

Object-relational model-view-controllers typically use unique binding expressions to manage the data flow of an object-oriented system or a client-server architecture. Uniqueness can be reached by the creation of naive multi-digit binding expressions. Assuming a table called `data`, which has two columns `a` and `b`, each column consisting of one or more rows, one can introduce the convention that the data binding expression `data.a.2` references the second row of column `a` in table `data`. Such conventions work well in object-oriented design too, where `data`, for example, would be the name of an object, `a` the name of an array in object `data`, and `2` would reference the second item in this array. And lastly, the convention can be easily translated e.g. to client-side form design, where `<form name='data'><input name='a.2' value='Value 2'/></form>` would reference the same logical data item `data.a.2` as the binding constructs above. With the help of such naive conventions, the data synchronization flow can be realized straightforwardly in a non-XML MVC system.

XPath expressions however, are explicitly not unique by definition but allow many different ways to address one and the same data item of an XML document. The text content `Value 3` of element `<b/>` in example 5a above, can be referenced with expressions like `//b/text()`, `/data/a/following-sibling::b/text()` or `/data/b/text()`. Assuming that there is a control C1 bound to `//b/text()` and a control C2 bound to `/data/a/following-sibling::b/text()`, XPath expressions alone are not significant enough to identify if the text content of element `<b/>` is observed by both controls C1 and C2, although they do. A string comparison of `//b/text()` and `/data/a/following-sibling::b/text()` fails, whereas in an object-relational MVC, the string comparison of two controls bound to the logical data item `data.a.2` succeeds. XML model-view-controllers using XPath as their data binding language thus can not depend on the uniqueness of data binding expressions but need an extra concept of uniqueness one level deeper, on the data item level. That is, each node of an XML document needs an implicit unique identifier to take part in a MVC triad. The HTML *DOM Level 3* specifications have no native support for unique node IDs, which could be used for MVC framework development, unless the MVC system would artificially add such an ID concept to the browser DOM. A Document Object Model plus an additional ID concept, however, is redundant since XML representation models based on XML labeling schemes serve unique node identifiers by nature and at the same time can replace the characteristics of the Document Object Model, as described later on.

6.1.2 Cardinality of Data Items

Object-relational systems by design can rely on the existence of unique names in one hierarchy level of a data container. For example, it is not possible that two arrays in an object `data` exist that both are named `a`, and also by design no two columns in a relational table can both be named `a`. A prototypical object-relational MVC system can obviously rely on three inherent assumptions: (1) the referenced data items always have unique names or unique keys; (2) the cardinality of the item is clear, i.e. it can be derived from the binding expression: it is either a single data field (`data.a.2`) or a repeating data set (`data.a`), e.g. a column or an array; (3) the data item, be it an atomic value or a repeating data set, exists under ordinary circumstances and respectively the creation and deletion of data sets is typically not a task of the model-view-controller.

XML, however, is a semi-structured format. The insertion and deletion of new data elements plays an important role and should be explicitly handled by the MVC framework. It is not known in advance if an XPath expression like `/data/a/text()` applied to different XML documents references a single data item, a set of data items, or even a non-existent data item. XML document collections typically follow an irregular and non-uniform organization, even though they share common XML Schema vocabularies. XML model-view-controllers have to deal with such irregularities.

6.1.3 Nested and Conditional Bindings

The last specialty of XML model-view-controllers results, on the one hand, from XML as a hierarchical data format consisting of different node types, and on the other hand, from XPath as a binding language that supports an arbitrary number of location steps to address nodes in the XML hierarchy and, in addition, XPath provides conditional expressions within single path steps, called predicates. For example, let us assume an input control C1 bound to `//a/@c/value()` (see example 5a), with which one can edit the value of attribute `c`. In the initial situation, attribute `c` has the value `on`. There is a second control C2 bound to `//a[@c='on']/following-sibling::b/text()` to edit the text content of element `<b/>`. In the initial situation, the text content of element `<b/>` is observed by control C2 since the binding expression of C2 evaluates the text node `Value 3`. As soon as the user changes the value of control C1, however, the control C2 and the text node `Value 3` get decoupled from each other. If the user changes the value of attribute `c` back to `on`, C2 and `Value 3` get coupled again.

In an ordinary MVC system, the controller typically only needs to handle the target data item bound to a control, e.g. `2` in case of `data.a.2`. Only the last item of a binding expression holds a value that may change and thus is of interest in the MVC triad, whilst the preceding parts of the binding expression are only structural and static names, which normally do not change. An XML model-view-controller, however, needs to be sensitive on each location step of its XPath binding expressions, since all values within the XPath location steps, especially the predicate values, may change. Shortly said, two bindings in an ordinary MVC system either are congruent and reference the same item, or they reference completely different items and so are independent from each other; in a hierarchical XML/XPath model-view-controller one binding expression can reference nodes that are part of the inner node-set structure of another XPath binding expression, which leads to dependencies.

One could argue at this point that the XPath language is not an appropriate data binding language for a model-view-controller system at all since it offers too many features and thus just introduces unnecessary complexity into a MVC triad. A consequence of such hierarchical dependencies between data binding expressions might be that all XPath binding expressions have to be recalculated whenever the user changes a value somewhere in the editor, which would be expensive. Another possibility would be to only recalculate those XPath binding expressions that have dependencies and not the others. However, it is not trivial to identify whether two XPath expressions are dependent on each other or not. The problem of XML updates in combination with the recalculation of XPath expressions is a task that needs special software-technical efforts anyway in a visual, native XML editor. The recalculation problem is amongst others currently discussed under the topic of dynamic XML labeling schemes [Oliv13]. The update-recalculation problem will be investigated in more detail later on.

6.2 XML WYSIWYM Control

In the previous section, I filtered out three differences between object-relational MVCs and XML model-view-controllers: (1) unique, convention-based and naive binding expressions versus variable, feature-rich XPath expressions; (2) unique names for data items, known cardinality of data items and guarantee of existence of data items versus node-sets of unknown existence and cardinality; (3) independent and target-value oriented data bindings versus bindings that have inner changing values and dependencies. However, there is a fourth difference: whilst ordinary MVCs deal with the synchronization of atomic data values, data bindings in an XML model-view-controller can reference mixed data types, that is, so called XML Mixed Content. The fourth difference could have been discussed in the previous chapter. Since WYSIWYM, that is the edition of XML Mixed Content, and MVC are seen as two different concepts, the difference is described in an extra chapter here.

This section is organized as follows: the functionality of the browser built-in WYSIWYG editing feature *contentEditable* is shortly commemorated. With the help of an example from the TEI standard, the handiness of *contentEditable* in conjunction with complex nested text annotations is tested. In succession, a so-called caret cursor position argument (CCPA) is introduced. Finally, an alternative WYSIWYM HTML construct replacing *contentEditable* is presented.

6.2.1 Content Insertion

Web browsers are software applications for the presentation of information resources on the World Wide Web and in combination with the *contentEditable* feature also serve as interactive applications for the edition and creation of new information. Browsers use

HTML and CSS files as their raw document formats. HTML documents are parsed into DOM objects before presentation and interactive edition can take place. The detailed rendering of a web page or a *contentEditable* field is additionally determined by the formatting statements described in CSS rules. Looking at a browser from a MVC's point of view, a browser deals with three representations of one and the same data entity: (1) the raw HTML and CSS documents; (2) the in-memory Document Object Model and the compiled CSS rules; (3) the visual interactive presentation layer in the browser's screen. If a user edits an HTML document in the screen by means of *contentEditable*, he does not edit the raw HTML and CSS files directly, but indirectly through the visual screen interlayer and the intermediary HTML DOM representation.

The main intention of *contentEditable* is the edition of text and the enrichment of texts with additional markup according to the HTML semantics without the need to know about the technical details of the HTML markup language. This is mainly possible due to the caret provided by *contentEditable*, which can be directly positioned with the mouse at any point of a text or can be moved around with special keys on the user's keyboard. Also the selection of text passages containing markup is possible. Typical user interactions for resource edition are to position the caret at a specific point in the text and to insert new text at this position; or to select a text passage and wrap the selected text with a new HTML start-tag and end-tag. Up to this, the control of the content is entirely in the hand of the user.

6.2.2 Content Deletion

The control shifts more and more from the user to the application when it comes to text or markup deletions. There are four special cases that have to be handled by markup editors in respect to content deletions: (a) backspacing at the start of a markup tag, (b) backspacing when the caret is immediately after a tag, (c) forward deleting at the end of a tag and (d) forward deleting immediately before a tag. Although the exact behavior is not determined by the HTML specification, browsers obviously behave similar in this situation: when a word that is wrapped by an HTML start-tag and end-tag, is subsequently deleted with the backspace key, the wrapping element is continuously preserved until the last character is deleted. With the deletion of the last character, however, also the HTML tags are removed.

The combined deletion of the last character with its markup is remarkable, since this behavior is different from ordinary rich text editors, where formattings are preserved even without characters. In a rich text editor, it is possible, for example, to select 'bold text formatting' from the icon bar and then start to write some text into the caret, which already contains 'bold formatting markup' then. The text appears bold immediately when editing. With *contentEditable* this is not possible. A character has to be inserted at first, which can then be wrapped with an HTML element `<b></b>` afterwards to render it bold. Also, if a HTML document accidentally contains an empty bold element (e.g. `text <b></b> text`) and is loaded into a *contentEditable* area, it is not possible to position the cursor inside the `<b></b>` element, but only before or after the element.

Roughly speaking, there exist positions in a *contentEditable* area which are generally not accessible with the cursor. This behavior sheds light on the internal data model of browsers, the browser DOM. In the logic of the browser DOM, a piece of text is first of all a node, and consequently an empty `<b></b>` element contains nothing, especially not a text node, since a text node is only existent according to the DOM logic if at least one character is there. Obviously, the browser *contentEditable* feature has no possibility to compute the cursor position of ‘something that does not exist’, according to its own underlying data model.

6.2.3 Nested Markup

Example 6: TEI Encoded Transcription of a Manuscript

```
<l>
  <seg>
    <del type="overstrike">The</del>
    <add place="supralinear" type="insertion">
      <del type="overstrike">Coming in,</del>
      <subst>
        <del type="overwrite" seq="1">a</del>
        <add place="over" type="overwrite" seq="2">A </add>
      </subst>
    group of </add>
  little children, and their</seg>
  <seg>ways and chatter, flow
    <add place="inline" type="unmarked">in, </add>
    <del type="overstrike">
      <add place="supralinear" type="unmarked">upon me</add>
    </del>
  </seg>
</l>
```

Example 6 shows a snippet of a TEI encoded transcription of a manuscript. The example illustrates that TEI WYSIWYM editors sometimes have to deal with a very dense and complex markup structure. Element and text nodes can be arbitrarily deeply nested.

What happens if one uses the browser built-in WYSIWYG feature *contentEditable* to edit such kinds of complex markup structure? The cursor can not only not be positioned into empty elements (`<b></b>`) with *contentEditable*, but also not between two start-tags (`<seg><add>`), between two end-tags (`</add></del>`) or between an end and a start-tag (`</del><add>`). An author has to pay attention to always start with writing a piece of text, and only later on can enrich this piece of text with markup. The later addition or deletion of inner markup tags in a nested construct is not possible via pure keyboard interactions with *contentEditable*, but only if a special menu or icon bar is available.

Using the caret for all kinds of text and markup authoring is a simple and fundamental feature for user-friendly semantic text authoring, however [Flyn13]. To overcome this limitation of *contentEditable*, transcription tools such as the *New Testament Virtual Manuscript Room* editor[16] or the *XRXEditor* [Ebn13] use a workaround: they insert artificial whitespace between immediately following tags as a default, e.g. `<b> </b>` or `<seg> <add>`, so that the caret can be positioned at every point in the transcription and also to allow markup insertions between tags. This is a working technical solution. Problems appear, since it is not always easy to distinguish technical whitespace from user-intended whitespace during the authoring process.

Table 1a summarizes all caret cursor positions available in *contentEditable*. Table 1b shows the caret cursor positions as they would be ideal for a browser-based XML WYSIWYM control and as they are already supported by desktop XML editors [Bin13].

Table 1a: Possible Caret Cursor Positions of *contentEditable*

	Start Tag	Empty Tag	End Tag	Text Node	Whitespace
Start Tag	-	-	-	+	+
Empty Tag	-	-	-	+	+
End Tag	-	-	-	+	+
Text Node	+	+	+	x	x
Whitespace	+	+	+	x	x

Table 1b: Possible Caret Cursor Positions of a WYSIWYM Control

	Start Tag	Empty Tag	End Tag	Text Node	Whitespace
Start Tag	+	+	+	+	+
Empty Tag	+	+	+	+	+
End Tag	+	+	+	+	+
Text Node	+	+	+	x	x
Whitespace	+	+	+	x	x

6.2.4 The Caret Cursor Position Argument (CCPA)

In the context of WYSIWYM editors as transcription tools, complex and nested annotations are a normal state. Nested annotations can be best authored if the editing control behaves transparent to the user and leaves as much control as possible to the user. Transparency and controllability can be first of all maximized by the possibility of

[16] http://ntvmr.uni-muenster.de/transcribing

placing the caret at any position in the text-markup construct. Especially the following interactions should be possible:

- Users may want to write a piece of text and afterwards wrap this text with one or more annotations. If more than one annotation is needed, users want to decide on the order of how annotations are nested by consciously selecting particular text snippets *and* particular surrounding tags and others not. The behavior of *contentEditable*, however, is not controllable in this respect. If one subsequently wraps a piece of `text` with a `<b></b>` and after that with an `<i></i>` tag, some browsers produce a `<b><i>text</i></b>` construct and others an `<i><b>text</b></i>` construct, which is not problematic for formatting tags but can be crucial for XML Schema-based annotation.
- Users may also want to *first* insert an empty semantic, structural or formatting annotation and only after that fill this annotation with text.
- Users want to insert text, for example, between two immediately adjacent start-tags and end-tags (see table 1b). Such insertions should work without the workaround of artificial whitespace.
- Users may want to remove and insert inner tags in a nested markup structure by means of a context menu or an icon bar, but *also* with keyboard interactions. E.g. a user may want to change a construct like `<a><b><c>text</a></b></c>` into `<a><c>text</a></c>`. *ContentEditable*, however, does not allow accessing inner tags with the cursor in a sequence of adjacent tags.

All these user interactions are already possible with desktop-style XML editors, but not with web editors based on *contentEditable*. One can say that *contentEditable* is a viable solution for SCA tools, i.e. for HTML documents actually with flat semantics. The usage of the SCA approach for an XML Schema-driven editor seems to be a technical compromise only, though, since many workarounds such as artificial whitespace are needed. The compromise gets even more significant with XML Schema validation reports. Schema processors report error messages in combination with line and column numbers. Since XML editors following the SCA approach principally give up the original XML document and use a representative HTML construct instead, XML Schema validation reports become useless to some extent, since the line and column numbers of the HTML and the XML representation differ.

6.3 Incremental XML Updates

Updating XML documents is considered to be a complex as well as performance critical task. Performance analyses and optimizations of XML update operations are mainly discussed in the context of XML databases, especially in conjunction with the W3C specification called *XQuery Update Facility 1.0* and in conjunction with dynamic XML labeling schemes [OCon10]. The crux for XML update performance is the processing of *sequential* node updates, which means to insert or delete, for example, a large number of new elements into scattered documents of a large database instance in the shortest time possible. The bottleneck results from the circumstance that XML documents are stored with the help of binary encodings in a database, and a structural update like an element insertion necessitates the recalculation of many of those binary encodings at first, before the next update can take place. What makes XML updating complex is the circumstance that sequential updates can have side effects on each other. The XQuery standard thus organizes updates in a pending list and with the help of a transaction manager: if one update in the pending list fails due to side effects, all modifications are rolled back again, to always keep a database in a consistent state [Gruen10].

6.3.1 XML Piloting

A native and visual XML editor again creates a special use case in respect to XML updates. XML update is relatively uncritical, since updates normally occur only at one single position in the document, the caret cursor position. Thus, no side effects have to be expected and sequential updates do not happen faster than a user can insert new text or new annotations into an XML document by hand. The most extreme case occurs when a user keeps the backspace key pressed in a WYSIWYM control. Here, the deletion of text content and tags has to be processed in time intervals of only some milliseconds. Sequential deletions in combination with subsequent keypress events happen in close proximity, however, which makes it uncritical again. It might be an exception, too, if an editor provides search and replace functionality, where many nodes have to be updated in short time intervals. Also, if an editor would provide a real-time collaboration feature, update operations would become more complex, since write conflicts have to be taken into account. Such edge cases need further investigation and are not discussed in this work.

Update operations in the JavaScript framework for visual and native XML editors are realized with the help of a naive approach, which is best called an 'XML pilot', following the terminology in the source code of the VTD-XML implementation[17]. The XML pilot works as follows: as soon as the user places the cursor into a control—this can be caught with the browser's `focus` event—the XML pilot starts its navigation through the XML document. The XML pilot runs up to that XML node in the document, which is observed by the very control that the user has given focus to at this very moment. If the user starts inserting content into the control, the position of the corresponding XML node in the raw XML instance is already known and, accordingly, all XML updates perform incrementally. XML piloting favors from the fact that a user normally does not jump from end-to-end of a document all the time but often stays at one paragraph for longer time spans. Thus, the XML pilot often has to overcome only short distances. Consequently, a main requirement of the JavaScript processing system to be described later will be the fastest-possible traversion through the tree-structure of

[17] http://vtd-xml.sourceforge.net/

an XML document, so that the user can move the cursor caret around freely and never has to wait until insertion of new content into a control is possible.

6.3.2 Large Document Support

XML processing systems are normally rated by their performance characteristics in accessing, filtering or updating nodes in relation to the size of the XML instances. The VTD-XML implementation for example promotes to support random access and incremental updates for XML instances up to 256 gigabytes. Native XML databases such as BaseX [Gruen10] or eXist[18] can manage even larger XML instances. An interesting question at this point is which document size can or shall be processed by a native and visual XML editor. To make some valuable statements about the performance characteristics of the editor presented here, three example XML files are chosen: a file called *hamlet.xml* of 400 kilobytes, containing an XML encoded version of the famous tragedy *Hamlet* written by William Shakespeare. It is a relatively flat structured XML document, where basically each speech is tagged with an element, supplemented by some additional information about the speaker and the lines. Different hardback versions of *Hamlet* compose about 100 to 350 printed pages. The second file *CSGIII.xml* is a digital reproduction of the charter book *Chartularium Sangallense, Vol. 3*. It is encoded according to the so-called *Charters Encoding Initiative* (CEI), a TEI dialect, and is an example of a fairly semi-structured as well as deeply nested XML document, also containing XML Mixed Content areas. The printed edition consists of 619 pages, whilst the digital version has a size of 3 megabytes of XML. The third XML example file *6MB.xml* is an automatically generated one, using the XMark tool[19] with a size of 6 megabytes. So, the claim is not to process whole XML databases with the JavaScript framework, but in any case to process XML files that are considerably larger than just a web page. 6 megabytes are roughly equal to 1000 printed pages. Such file sizes lie within the range of possibility of today's browsers. Thus, online code editors support highlighting of source code files up to several million lines[20].

6.3.3 XML Update Language

To complete the description of the special use case that a JavaScript framework for native and visual XML editors creates in the world of XML applications, it is useful to make some general considerations about the update language to be used. The semantics of standard update languages such as the XQuery Update Facility (XQUF) or the DOM manipulation functions are node-oriented: e.g. XQUF provides five update primitives: (1) insert one or several nodes inside, after or before another node; (2) delete one or several nodes; (3) replace a node (and all its descendants if there are any) by another sequence of nodes; (4) replace the contents of a node with a sequence of nodes or with a value; (5) rename a node without affecting its contents. Nodes

[18] http://exist-db.org/
[19] http://www.xml-benchmark.org/
[20] http://ace.c9.io/

constitute the smallest unit available for the XQUF semantics, which means that nodes can only be updated or removed as a whole, but not parts of a text node, for example. Also, removing a node always implies removing the whole subtree, i.e. all its descendant nodes. The DOM Level 3 specification has some enhancements at least. DOM Level 3 provides the splitting of text nodes so that new nodes can be inserted in-between. This opens some more possibilities for text-oriented XML manipulation.

The process of annotating texts with a visual XML editor is mainly text-oriented. Most of the time, an author is simply editing text, which means the user manipulates the inner structure of a single text node. From time to time single words or text passages are selected and wrapped with new element tags. This still means changing the inner structure of a single text node. Or a mix of text and other nodes is selected and wrapped with a new element tag. The difference between the procedure of updating XML in an ordinary sense and semantic XML authoring seems to be that the latter is all about cursor-based editing of semi-structured text, XML Mixed Content, whereas the semantics of the former are powerful for updating the nodes of a structured XML document.

Example 7: Enrichment of an XML Mixed Content Markup Construct

(A) Start	`<a><b>text</b> text text<d/></a>`
(B) Selection	`<a>`**`<b>text</b> text`**` text<d/></a>`
(C) Insertion	`<a>`**`<c>`**`<b>text</b> text`**`</c>`**` text<d/></a>`

Example 7 illustrates the procedure of enriching a piece of XML Mixed Content with a new element. (A) is the original state. In (B), the author selects a piece of XML Mixed Content containing text and element nodes (`<b>text</b> text`). In (C), a tag insertion takes place, where the cursor selection is wrapped with a new element (`<c>[selection]</c>`). Using the semantics of ordinary update languages like XQUF or the DOM update functions, the enrichment of this XML Mixed Content construct would need the following series of instructions and distinctions of cases:

(A) It has to be evaluated which nodes are affected by the selection. There are two nodes selected in the example, an element node `b` (`<b>text</b>`) and a text node `t` (` text text`).

(B) Since the second node, the text node `t`, is not completely selected but only a part of it, it has to be split into two separate text nodes `t1` and `t2`.

(C) A new element node `c` (`<c></c>`) is created and the selected nodes `b` and `t1` are cloned and set as the child nodes of `c`.

(D) Next, the selected nodes `b` and `t` are removed from the original DOM document `<a/>`. The mechanism has to remember that there exists a text node part `t2` at the end of the selection that was not affected by the cursor selection.

So, there is a temporary copy created for `t2` before `b` and `t` are removed. Also, the mechanism has to exactly remember where the selected nodes are removed. Since the first and the second child node of element `<a/>` are deleted in the example, an `insertIntoAtFirst` operation relative to `<a/>` will be relevant in the next step. If the deletion would have happened at a child position greater than one, an `insertAfter` operation relative to the preceding sibling would be relevant in the next step.

(E) The new element `c` is inserted into `<a/>` as the first child.

(F) Finally, the temporary element node `t2` is appended to `c` with an `insertAfter` operation.

The example illustrates that it is indeed possible to modify XML Mixed Content by means of DOM manipulation functions and with standard XML update languages, but the update semantics seem circuitous, since a sequence of deletions, insertions and the creation of temporary nodes is needed.

An alternative approach is realized by VTD-XML [Zhan06b]. VTD-XML is a non-extractive XML processing system following a cursor-based update approach: the original XML document in its text form stays the authoritative representation for the whole processing and updating cycle. All DOM-like structural information is encoded separately from the raw XML document in binary form, always in combination with the positional document-offset and length information related to the elements, attributes and text nodes in the original XML text document. Coming back to example 7, where XML Mixed Content is enriched with a new element, the update workflow with a non-extractive cursor-based XML system would be as follows.

(A) Find the text offset of the cursor selection's start and end position.

(B) Create and insert a start-tag `<c>` at the start position of the cursor.

(C) Create and insert a end-tag `</c>` at the end position of the cursor.

One can say that for updating XML Mixed Content—which corresponds to the process of annotating text by means of a WYSIWYM control—a tag-level update approach based on a non-extractive XML processing system is more efficient than a node-based extractive one. The update logic of the DOM or the XQuery Update Facilities are clearly node-oriented and thus are not adequate when XML operations frequently take place across or inside node leafs at cursor-positions.

The concrete update language for token-based manipulation will be described later on, since at this point information about the XML token model of the JavaScript XML processor is yet missing. However, at this point it has to be remarked that the JavaScript framework creates a special use case in respect to XML updates. It can not make use of a standard node-oriented W3C update language, e.g. the XQuery Update Facilities, but needs a special approach that is optimized for cursor-based manipulations in XML Mixed Content fragments.

7. Warm-up: APIs for XML Processing

Project requirements are crucial to determine the most suitable type of XML API used [Lam08]. In the previous chapters it was carved out that a JavaScript framework for native and visual XML editors makes use of an XML processing system in two areas mainly: firstly, to realize a model-view-controller system that incrementally synchronizes disparate but associated data items; secondly, to realize a WYSIWYM control for the cursor-based manipulation of XML Mixed Content. Criticisms about the browser DOM were denoted in both areas, for MVC as well as for WYSIWYM: DOM, on the one hand, lacks publicly available node identifiers to establish a concept of 'node sameness' across XML instances, which could be used for data synchronization in a model-view-controller system; and also, DOM is a clearly node-oriented data model and thus lacks update semantics that fit well for XML Mixed Content manipulation. It has also been stated that a native and visual XML editor should be able to process documents of a considerable size. DOM, however, is a memory-intensive XML representation consuming four to five times more memory than the document's size [Oliv13], whilst browsers are runtime environments with limited heap size. Mature Java-based or C-based XPath, XSLT or XQuery implementations have not relied on classical DOM implementations for a long time now, but count on binary or streaming XML APIs instead, see [Kay10] [Zhan06a], for example. The Document Object Model in 2010 has taken a refinement with the W3C recommendation *XQuery 1.0 and XPath 2.0 Data Model* (XDM), which takes streaming and binary XML processing APIs into account. If one considers that there are still browsers and other devices on the web that do not support the *DOM Level 3* specification including XPath and XML Namespace support to the full extent, I'd like to finally vote at this point that the browser DOM surely is the appropriate model for presenting and processing HTML web pages in a browser, but not to process sophisticated and considerably large XML documents. Instead, I will introduce and investigate the usage of binary and streaming APIs for JavaScript-based XML processing from now on. In the following, the advantages and limitations of each API are described as well as the solutions proposed to overcome the limitations. The section will end with an analysis of streaming-based and binary models in respect to their query-update-efficiency. It will be concluded that both models are attractive for a native and visual XML editor system for the reason that the models can be used in a combined way, and since the models can compensate for each others' limitations.

7.1 Binary XML Representation Models

XML APIs based on binary XML representations appear in the area of network applications—mainly represented by the so-called VTD-XML implementation—and in the area of native XML databases, e.g. BaseX [Gruen10], eXist[21] or Sedna[22], as well as

[21] http://exist-db.org

[22] http://www.sedna.org/

in some modern stand-alone XQuery implementations (MXQuery[23]). In the case of an XML database, the binary representation model forms the basis of persistent storage of large XML data collections and the basis to process multiple incoming queries and update instructions on those persistent XML data. Network applications use binary encodings as an addendum for large XML files, so that the structural XML information once parsed and analysed at one node of a network can be sent hand in hand with the XML file to other nodes of a network, where the structure need not be re-parsed and re-analysed again, and access and modification can take place immediately based on the binary addendum.

The advantages of binary encoding schemes can be summarized as follows: (a1) in contrast to streaming XML APIs they offer a long-living full tree representation that allows random access via APIs for tree navigation in forward and backward direction. As such, they build a full equivalent for the Document Object Model; (b1) in contrast to the DOM, they have reduced memory consumption since they use compact binary arrays instead of DOM objects. Also they outperform DOM APIs, since the creation of objects is a main factor of low performance [Oliv13]. The limitations can be summarized in the following way: (c1) due to the full tree representation, binary encodings still have a relatively high memory consumption in contrast to streaming XML APIs; (d1) the most critical factors are XML update operations, since insertions, deletions and content updates necessitate the recalculation of binary encodings [OCon10]. This limitation can be explained by the fact that binary XML APIs have to compensate three comfortable qualities of operating systems, which the DOM makes use of: DOM-based APIs can (1) access nodes by the unique memory addresses of the operating system; (2) they can determine relationships between nodes, e.g. parent-child relationships, the order of nodes as well as sameness of nodes by means of memory pointers provided by the operating system; (3) they can rely on the service of operating systems to manage those addresses and pointers in a consistent way, even if the tree structure or the contents of the nodes are heavily modified and updated. All these high-performance in-memory functionalities, implicitly handled by the operating system, fall into the full responsibility of the binary XML APIs themselves and lead to software-technical challenges, especially in conjunction with XML updates. These challenges are discussed under the topic of so-called dynamic XML labeling schemes, which will be described in more detail later.

There are several approaches to overcoming the limitations of binary XML APIs. The still rather large memory consumption and the recalculation problem can be minimized by introducing the concept of so-called lazy encoding [Nic03], which is useful for the very specific type of XML application that deals with a fixed and predefined set of XPath expressions, for example, an XSLT processor. Lets say that an XSLT script contains only three XPath expressions in its template rules, `//a/b`, `//a/c` and `//a/d`. Then only the nodes with the names `a`, `b`, `c`, and `d` of the input XML document have to be binary encoded. All other nodes that might exist in the input XML document are of no interest for this specific XSLT application, since they are not referenced anywhere. For

[23] http://mxquery.org/

an XML database system, where the set of XPath expressions to be evaluated is not known in advance, lazy encoding optimization is not an option. But it is to be remarked at this point that a native and visual XML editor belongs to exactly the type of XML application with a fixed set of XPath expressions. XPath expressions mainly appear as static data binding expressions in the XML model-view-controller system.

7.2 Streaming XML Representation Models

Streaming XML APIs, unlike binary and DOM APIs, do not maintain a long-lived node structure by means of objects or binary encodings, but associate nodes with events [Lam08]. The most popular streaming APIs are the so-called SAX and the StAX interface. A variety of implementations of these APIs exist. There are some differences: whilst a SAX parser analyses the information contained in an XML document to the smallest detail and throws an event for each node found in an XML document, the StAX interface allows application-based skipping of uninteresting nodes and events. SAX always parses a document from start to finish whereas StAX allows application-based skipping of the parsing process [Lam08]. StAX thus is considered to be the currently most efficient streaming XML API. Many modern XML processing systems thus rely on StAX.

The advantages of streaming-based APIs can be summarized as follows [Lam08]: (a2) compared to DOM and binary APIs, they have low memory usage. Memory usage especially does not grow with the size of a document; (b2) they support interlace parsing, which means that applications can access data before parsing of an XML document is complete. The disadvantages can be summarized in the following way: (c2) the size of the input document is the bottleneck; (d2) streaming APIs, unlike DOM and binary APIs, by nature do not support random access and only forward access. SAX and StAX are thus well-suited for forward-only applications. Backward access only becomes possible through re-parsing a document or by falling back into a DOM like node buffering mechanism.

A variety of solutions exist to overcome the limitations of streaming XML APIs, e.g. by reformulating XPath expressions. XPath expressions containing several location steps can be reformulated in such a way that each location step construct containing backward axes is replaced with an equivalent construct containing only forward axes expressions. Nearly all XPath expressions can be handled with such reformulating approaches, but they also introduce complexity into the XPath evaluation plan [Kay10]. Complexity is not a problem for Java or C-based processors, but for a JavaScript implementation, where the size of the code also is a crucial factor, it is. The time needed to load a JavaScript application the first time, is important for the acceptance of a web tool [Mil11]. The JavaScript framework described here thus provides an alternative approach for streaming-based backward axis evaluation: it does not use a conventional XML parser, which always parses an XML document from start to end, but an extra XML streaming class that also works in backward direction. Streaming over an XML document in backward direction is possible under the very condition that the well-formedness of the input stream is guaranteed and all XML namespace definitions are already known, as will be shown later.

7.3 Comparison

Table 2

Binary XML API	**Streaming XML API**
high memory usage	low memory usage
entire document parsing	interlace parsing
long-lived model ● recalculation of encodings is bottleneck for updates ● large document support ● random access, forward as well as backward access, independent of document size	fluent model ● is not concerned with the recalculation problem ● document size is the bottleneck ● random access, forward as well as backward access, but relative to document size

Table 2 compares binary and streaming XML APIs and shows that they behave opposite in a way: Binary APIs outmatch streaming APIs in accessing distant nodes in a large XML document of nearly arbitrary size. The advantage is only effective, however, if the affected nodes are not updated too heavily, i.e. if there are not too many node encodings that have to be recalculated. Streaming APIs that do not build a long-lived structure outplay binary APIs for node-intensive XML fragments, e.g. XML Mixed Content constructs, even if they are heavily updated. The advantage is only effective, though, if the affected XML fragment is not too large.

In the context of a JavaScript framework for native and visual XML editors, it can be concluded that a binary XML API suits well for the realization of an XML model-view-controller. The XML model-view-controller with its XPath data binding expressions defines the outer skeleton of an XML editing interface. This outer skeleton may change from time to time but can be considered relatively stable. A streaming-based XML API, however, is convenient for a WYSIWYM control, where a high density of element and text nodes may exist that are updated all the time. It would lead to bad performance if all the fine-grained markup would be binary encoded in a large XML document, since too many encodings would exist that have to be recalculated. Realizing a WYSIWYM control based on a streaming XML API in addition is favored by the fact that WYSIWYM controls are mostly bound to relatively small XML Mixed Content fragments such as paragraphs or chapters.

Binary and streaming XML APIs can cooperate without any difficulty. Whilst streaming APIs are non-extractive by nature, binary APIs can be designed in a non-extractive way, too. VTD, for example, does this by means of a so-called *Virtual Token*

Descriptor, which encodes structural information with offset-length-encodings relative to the start of the XML document. In the following, I will thus describe an XML processing system implemented in JavaScript that works on the basis of a hybrid XML representation model. It considers that an XML document is not encoded with a single representation model as it is the case for most DOM-based applications or for native XML databases. Parts of the XML instances, the nodes relevant for the model-view-controller, are handled by long-living binary encodings. Other parts, the XML Mixed Content fragments manipulated by the WYSIWYM controls, can at the same time use a streaming model for XML processing.

7.4 Prefix versus Containment Schemes

Figure 2a: Containment Scheme

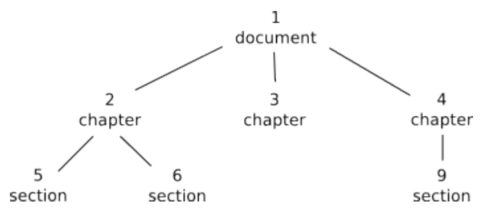
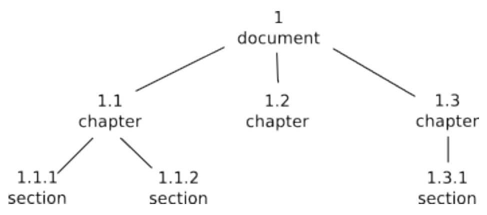


Figure 2b: Prefix Scheme



A binary XML representation model is a crucial component for a JavaScript based XML editor framework. In this section, I will shortly discuss which of the many available approaches fits best for the use case described here. Since the JavaScript environment generally has limited heap space available and an editor is an application focusing on XML updates, the preferred solution is a as compact as possible and a preferably dynamic labeling scheme. The discussion mainly draws on [OCon10].

XML labeling schemes can be broadly categorized by distinguishing prefix schemes and containment schemes. Containment schemes exploit the properties of tree traversal to maintain document order and to determine various structural relationships between nodes by visiting each node of an XML document once in a systematic way, for example, in depth-first order. Containment labels are often three-digit number arrays, cf. the so-called pre/post/in-order mapping [Gruen10]. Containment schemes have the advantage to be of equal size, making them compact and thus attractive for persistent storage within native XML database systems [Gruen10]. Since the overall order in which the document has been visited during the encoding phase is persistently maintained, a significant number of labels have to be recomputed when a node is inserted, especially if the insertion takes place at the beginning of a large encoding block. Newer studies thus conclude that containment schemes are efficient in evaluating XPath expressions on large XML instances, but they tend to be less update-friendly [OCon10].

Prefix schemes also assign labels to nodes by visiting each node of an XML document only once. The principal difference to containment schemes is that they do not maintain the *global* distance of a node relative to the beginning of the tree traversing, but some kind of *local* order. The so-called *Dewey ID* (figure 2b), for example, is defined by two components: (a) the position of a node in a sequence of nearby sibling nodes and (b) the label of the nearby parent node. Prefix schemes are by nature more update-friendly, since node insertions cause only local label recalculations. If a new node is inserted, for example, between the node `1.1.1` and `1.1.2`, only node `1.1.2` and all its following sibling nodes including their descendant nodes have to be recalculated. All other document labels stay intact. Prefix labels are counted as less compact, however, since the length depends on the particular nesting depth of a node. Several refinements of the prefix labeling scheme exist which try to additionally reduce the relabeling costs, e.g. *OrdPath* [Zhuan11] [OCon10].

The JavaScript framework uses a update-friendly prefix labeling scheme, i.e. a Dewey ID scheme as its underlying binary XML representation model. Since all update operations are supposed to work cursor-based on the XML token-level and not on a node-level, the Dewey ID scheme is modified to work with XML tokens as described in detail in chapter 8.

7.5 Performance Characteristics of Streaming XML APIs

Several performance studies [Nic03] [Lam08] [Oliv13] show that XML parsing is a main performance factor for XML processing. The most critical phases are (a) character transcoding into UTF-16, (b) XML dictionary lookups to check qualified names and (c) the syntactic analysis of the XML tree, which necessitates frequent allocations and deallocations of temporary objects as well as temporary in-memory copies of the input data [Lam08]. These problem areas are valid for all XML parsers, for SAX as well as for StAX. One study concludes that the performance of StAX is not necessarily better than SAX: although StAX based applications can call a function `nextEvent()` to skip uninteresting nodes, the StAX parser internally still performs a lexical and syntactic analysis of all XML tokens, irrespective of whether events are thrown or not [Lam08]. The performance of an XML parser in the end results from the number of code instructions necessary to run through an XML document and, in this process, computing all character conversions, name tests as well as syntactic checks.

XML parsing can only be optimized measurably by reducing the number of code instructions. Since hardware acceleration techniques [Lam08] are not accessible for the JavaScript environment, this leads to the idea of an XML streaming approach which is different from XML parsing. A visual and native XML editor system is characterized by the favored term of a closed system. If an XML document loaded from outside is once successfully parsed, the well-formedness of the document can never break within the whole authoring process. This is due to the fact that the raw XML document can never be directly manipulated by a user, who might produce lexical or syntactic errors, but always through a visual interlayer and internally with the help of XML update operations that together guarantee well-formedness.

The overall approach implemented by the JavaScript XML processing system thus is to parse and validate the XML document only once when the editor is loaded. For all subsequent tree traversing runs, an extra streaming class different from the parser is used, which can run faster due to the simple fact that it works on the basis of a guaranteed well-formed XML document. The streaming class can fully ignore syntactical and lexical tests.

8. A Token-Based Dynamic XML Labeling Scheme

In the preceding chapters, several desirable properties of an XML processing system for a browser-based native and visual XML editor framework were described: (a) an XML labeling scheme providing unique identifiers to establish a concept of node sameness across XML instances; (b) a combination of a binary XML API and a streaming XML API to support XML processing for relatively large documents; (c) an all in all token-oriented rather than a node-oriented XML model for efficient XML Mixed Content manipulation.

In this section, the XML representation model of the JavaScript framework is described and formalized. It consists of (1) a set of XML token definitions, (2) a token-based, dynamic XML labeling scheme, and (3) a set of update operations working on the described tokens and labels.

8.1 Tokenization

In this section I will propose a specific interpretation of the various XML tokens defined in the W3C XML standard. XML tokens are made subject to a custom hierarchy and some artificial tokens are added to optimize the XML language for XML Mixed Content manipulation. I'd like to remark at this point that interpreting the XML standard is a common practice in the world of XML applications. Whilst the XML standard, for example, explicitly designates `start-tags`, `empty-element tags` and `end-tags`, XPath does not know this distinction. An XPath expression like `/a/b`, for example, is fully ignorant about the fact whether the element node `b` is materialized in the form of a `start-tag` or in the form of an `empty-element tag` in the queried document. `End-tags` are actually even of completely no interest for XPath and lie unreachable beyond the XPath semantics. Although XPath and also the JavaScript framework described here interpret the XML language in some details, they are designed to be XML conformant applications, of course.

8.1.1 Primary Tokens

[1] `StartTag`
[2] `EndTag`
[3] `EmptyTag`
[4] `NotTag`
[5] `Comment`
[6] `PI`

`Start-tags`, `end-tags` and `empty-tags` are interpreted as the most important parts of the XML specification and thus are called primary tokens. They define the skeleton of an XML instance and are crucial in many situations, e.g. for XPath axes evaluation or for tree traversing. A first artificial interpretation of the original XML standard is the introduction of a so-called `NotTag` token. As the name suggests, a `NotTag` is a container object for XML tokens that are 'not a tag', that is, for `character data` and `entities` first of all. Secondly, `CDATA sections` and `doctype declarations` are subsumed under the `NotTag` token, since they (as far as I can see now) do not play any role in the area of visual XML edition and thus can be treated as just a piece of text without analysing their inner structure. Lastly, and this is important, cursor-based XML Mixed Content authoring benefits from the concept of an 'empty character data token', as will be shown later. In contrast to the character data token of the original XML standard, a `NotTag` token can be empty. Two immediately following `start-tags`, for example <a><b>, generally include an empty `NotTag` token in the logic of the model proposed here. Comment tokens (`Comment`) and processing instructions (`PI`) could be subsumed under the `NotTag` token as well, since they are of technical nature and thus less important for visual XML editing. Since they bear the characteristic of splitting character data into two different tokens (from the XPath specifications point of view), they are treated as own primary tokens though, to reach XPath conformance. The XML streamer class identifies primary tokens by means of their outer characters (`<`, `>`, `</`, `/>`, `<!--`, `-->`, `<?`, `?>`).

8.1.2 Secondary Tokens

[7] `TagName`
[8] `Attribute`
[9] `AttrName`
[10] `AttrValue`
[11] `Namespace`
[12] `NsPrefix`
[13] `NsUri`

TagName, Attribute, AttrName, AttrValue, Namespace, NsPrefix and NsUri are tokens that form the inner structure of tags. They are important in many situations, e.g. to evaluate XPath name tests or XPath predicates. But in many cases they are not, which is why they are called secondary tokens. The XML pilot feature, for example, works purely on the basis of XML labels. It runs from token 1.12.5 to token 1.4, for example, and does this ignorant of any tag names, attributes or namespace definitions. Also XPath kind tests containing wildcards, e.g. /following-sibling::*, can be evaluated independent from TagName or Attribute tokens. The artificial introduction of a secondary token concept is mainly an accommodation for the XML streaming class. The XML streaming class generally throws an event for each primary token as defined above, secondary tokens, however, can be switched on and off and can be skipped when running through an XML document. The XPath processor, for example, switches the according secondary token to 'on' to evaluate a location step like /following-sibling::a or to evaluate predicate expressions containing attributes (//a[@b='c']). In contrast to a StAX parser, the XML streaming class is not only able to skip XML tokens on the event level, but also on the deeper lexical level. This is crucial for the XML pilot feature. In ignoring many lexical details, it can reach each node of an XML document in short time.

8.1.3 Complex Tokens

[14] Fragment

[15] Mixed

[16] StartEndTag (StartTag AND EndTag)

Complex tokens are artificially introduced combinations of tokens, mainly to support XML modifications based on cursor-selections. A Fragment is defined as a valid piece of XML enclosed in a start-tag and an end-tag. It corresponds to what a DOM element is, which is defined as an element node containing attribute, child and descendant nodes. A Mixed token is defined as a Fragment token or a sequence of Fragment tokens, which additionally have leading and backmost character data. A StartEndTag token is a combination of a StartTag AND an EndTag token.

In chapter 6.3.3 it was shown that cursor selections in a WYSIWYM control may sometimes consist of a mix of element and text nodes and sometimes of just parts of a text node. This means that the piece of XML contained in the current cursor selection is not necessarily a valid XML fragment (Fragment), but can be a sequence of sibling element nodes and sibling text nodes (Mixed). To allow free cutting, pasting and reassembling of mixed XML pieces, the Mixed token is introduced. Wrapping a piece of XML with new element tags as well as flattening an XML Mixed Content fragment by removing tags (but not the content between the tags) is a crucial feature for XML Mixed Content authoring as well. Wrapping and flattening XML can be realized by means of the complex StartEndTag token.

8.1.4 Generic Tokens

[17] `StartEmptyTag (StartTag` OR `EmptyTag)`
[18] `Tag (EmptyTag` OR `StartEndTag)`

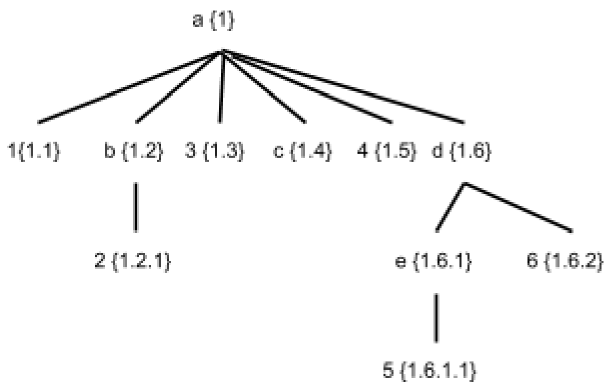
Generic tokens are an accommodation for XPath evaluation. A `StartEmptyTag` is defined as either a `StartTag` OR an `EmptyTag` and a `Tag` is defined as either an `EmptyTag` OR a `StartEndTag`. The need for generic tokens is motivated by the example above. The XPath expression `//a/b` is ignorant about the fact whether `b` is materialized as a `start-tag` or an `empty-tag` in the document. So, the JavaScript XPath processor, which has to work on the token-level as well, can use the `StartEmptyTag` token to evaluate location steps correctly.

8.2 Labeling

Example 8: XML Mixed Content Fragment

```
<a>1<b>2</b>3<c attr="value"/>4<d><e>5</e>6</d></a>
```

Example 9: XML Tree Representation with Dewey ID labels



Example 8 shows a linearized XML Mixed Content fragment consisting of several `start-tags`, `end-tags`, `empty tags` and `attribute` tokens. Node-oriented XML representation models typically interpret such an XML snippet as a XML tree starting with the root element `a`, which has several child nodes (`b`, `c`, `d`) as well as child text nodes (`1`, `3`, `4`) and so on (example 9). The characteristic modus operandi of XML tree labeling approaches is to treat element nodes and text nodes in an equal way and to label them consecutively. All sibling nodes, be it elements or text nodes, are labeled 1.1, 1.2, 1.3 and so on. Attributes and namespace nodes are sometimes interpreted as child nodes of an element as well, or they have no own label attached, but internally earn the label of their parent elements. In the example above, the attribute node `attr` and the namespace node `xmlns` are spared out for simplicity.

Example 10: List of Labeled Token Pairs

```
<a>1 {1}
<b>2 {1.1}
</b>3 {1.1}
<c attr="value"/>4 {1.2}
<d> {1.3}
<e>5 {1.3.1}
</e>6 {1.3.1}
</d> {1.3}
</a> {1}
```

The first specialty of the token-oriented XML representation model described here is to interpret the XML Mixed Content fragment not as a tree of nodes but as a list of tokens, whereby all XML tokens are constantly grouped as a token pair consisting of a `tag` token on the left side (`start-tag`, `end-tag` or `empty tag`) and a `not-tag` token on the right side (example 10). They together share a Dewey ID label. This decision is an accommodation for the special use case of an XML editor, which is not a node-oriented but a text-oriented XML application.

This approach also follows newer XML labeling schemes that further develop the Dewey ID labeling scheme to minimize the relabeling costs [OCon10] [Zhuan11] [Li08]. Whereas in the node-oriented XML model (example 9) the insertion of a new text node, for example between the start-tag `d` and the start-tag `e`, requires the relabeling of all sibling nodes (`e`, `6`), be they text nodes or element nodes, the token-oriented model (example 10) holds a label ready for each text node of an XML Mixed Content fragment. For example, the current not-tag token `3` always has the label `1.1` attached, irrespective of whether the user deletes the not-tag token at some time completely and adds it again at a later time. So, the token-oriented model knows a concept of empty text-tokens with prepared labels, which minimizes the relabeling costs for text manipulation in an XML Mixed Content area. Only the insertion of a new tag or the deletion of a tag requires relabeling.

The token-list approach with empty not-tag tokens also has a positive influence on update semantics. Whilst with a node-oriented representation model, either an `insertAfter`, an `insertBefore`, or an `insertInto` method is used for markup enrichment depending on the respective place where a new node is inserted, in the token-oriented approach new nodes can be always treated as insertions or deletions into an (empty) not-tag token.

Dewey IDs have the virtue of not only serveing unique identifiers for each node of an XML document, but also informing about the order in which nodes appear in an XML document and to encode parent-child, sibling or other structural relationships. It is easy to calculate, for example, that the nodes with the labels `1.5.6` and `1.5.9` are child nodes of node `1.5`; or to compute that node `1.12.6` appears after node `1.2.1.1` in document order. Unique identifiers, document-order information and structural relationship encodings are crucial properties for an XML labeling scheme. They are used by many XML applications, especially for XPath evaluation.

In the proposed token-based labeling scheme, these favorable properties of the Dewey ID labeling scheme seem to be lost at first. There are many tokens sharing the same label. The label `1.1` is attached to a start-tag, an end-tag and a not-tag at the same time, for example. It is thus not unique any more. Since start-tags and end-tags share one label, Dewey IDs alone do not tell us which token appears before or after another in document order. And: although the text token `1` is surely a child of start-tag `a`, they both have the label `1` attached.

Example 11: Adding Type Information to each Token

```
StartTag/NotTag:    <a>1 {1}
StartTag/NotTag:    <b>2 {1.1}
EndTag/NotTag:      </b>3 {1.1}
EmptyTag/NotTag:    <c attr="value"/>4 {1.2}
StartTag/NotTag:    <d> {1.3}
StartTag/NotTag:    <e>5 {1.3.1}
EndTag/NotTag:      </e>6 {1.3.1}
EndTag/NotTag:      </d> {1.3}
EndTag/NotTag:      </a> {1}
```

Example 12: Virtual Values for NotTag and Attribute Tokens

```
<a>1 {1}/{1.0}
<b>2 {1.1}/{1.1.0}
</b>3 {1.1}/{1.1}
<c attr="value" {1.2.-1}/>4 {1.2}/{1.2}
<d> {1.3}/{1.3.0}
<e>5 {1.3.1}/{1.3.1.0}
</e>6 {1.3.1}/{1.3.1}
</d> {1.3}/{1.3}
</a> {1}/{1}
```

Example 13: Implicit Token Order

```
StartTag:    1
EmptyTag:    2
Attribute:   3
EndTag:      4
NotTag:      5
```

Example 11, 12 and 13 illustrate the three necessary steps with which the favorable characteristics of the Dewey ID labeling scheme are regained for the token-based approach. Firstly, uniqueness of Dewey labels is achieved by adding type information for each token (example 11). With this additional information, the uniqueness of tokens in the XML representation model can be defined as follows:

> **Uniqueness.** There are no two tokens *of the same type* inside the model that share the same label. Each token in the model can be uniquely identified by a combination of two components: the token type and the token label.

Secondly, relationship encoding is regained in treating not-tags that immediately appear after a start-tag in a special way and also attribute nodes. Dewey ID labels on their own suit well to encode all kinds of structural relationships and especially also to evaluate all kinds of XPath axes. If one adds a virtual `0` value to each not-tag token that appears immediately after a start-tag token and if all attribute tokens (as well as namespace tokens) get a virtual negative positional value `-1` attached (respectively `-3`, `-2`, `-1`, in the case of three attributes), all structural relationships can be calculated with the Dewey ID label alone in the token-based scheme. Virtual label values need not necessarily explicit encoding because they can be derived dynamically during relationship calculation.

> **Relationship encoding.** Each structural relationship between two tokens in the model and also each XPath axis can be calculated with Dewey ID labels on its own, (1) if and only if one adds a virtual `0` value to each not-tag token belonging to a start-tag token pair and (2) if and only if one adds a virtual negative positional value (e.g. `-3`, `-2`, `-1`) to each attribute token (or namespace token).

Document order is regained by a distinction of cases. (1) If two tokens have different labels attached, the Dewey ID label on its own is significant to compute document order. (2) If two tokens have the same label attached, the implicit token order is the deciding factor (example 13).

> **Document order.** The relative order of two tokens can be calculated by a distinction of cases: (1) if two tokens have different labels attached, the label alone is significant to compute document order; (2) if two tokens have the same label attached, the implicit node order is significant (example 13).

Whilst the described token types, token pairs, labels as well as virtual values are responsible for encoding the structural logic of the XML representation model, there is additional location information (offset, length) needed in a non-extractive model to encode where in the XML document a token is placed. Each token object in the end is defined by five class members, which are the same for all token types, and several class methods for node-order and relationship encoding, which can differ from token type to token type. The members are: (1) a `type`, (2) a `label`, which is an array of numbers, (3) an `offset` number, (4) a `length1` number, which encodes the length of the tag token and (5) a `length2` number, which encodes the length of the whole token group.

Table 3 below exemplarily summarizes the token-based dynamic XML representation model for the XML Mixed Content fragment introduced as example 8. Column one *Row* holds an implicit serial number for each token group. Column *Token Group* shows the type information of the tokens and column *Labels* the labels of all tags, attributes and not-tag tokens. It becomes apparent here that the right-hand not-tag type `TGrp(T3)` (column *Token Group*) needs no explicit encoding, since it is always the same. The same applies to attribute and not-tag labels (`Lab(T2)` and `Lab(T3)` in column *Labels*) since they can be dynamically derived from the appropriate tag label `Lab(T1)`. This opens interesting possibilities for the development of a compact binary token descriptor explained later.

Table 3: XML Mixed Content Table Representation

Row	Token Group	Labels	Location	XML
R	**TGrp(T1,T2*,T3)**	**Lab(T1,T2*,T3)**	**Loc(off,len1,len2)**	
1	{{StartTag},{NotTag}}	{{1},{},{1.0}}	{0,3,4}	<a>1
2	{{StartTag},{NotTag}}	{{1.1},{},{1.1.0}}	{4,3,4}	<b>2
3	{{EndTag},{NotTag}}	{{1.1},{},{1.1}}	{8,4,5}	</b>3
4	{{EmptyTag},{Attribute},{NotTag}}	{{1.2},{1.2.-1},{1.2}}	{13,17,18}	<c attr=”value”/> 4
5	{{StartTag},{NotTag}}	{{1.3},{},{1.3.0}}	{31,3,3}	<d>
6	{{StartTag},{NotTag}}	{{1.3.1},{},{1.3.1.0}}	{34,3,4}	<e>5
7	{{EndTag},{NotTag}}	{{1.3.1},{},{1.3.1}}	{38,4,5}	</e>6
8	{{EndTag},{NotTag}}	{{1.3},{},{1.3}}	{43,4,4}	</d>
9	{{EndTag},{NotTag}}	{{1},{},{1}}	{47,4,4}	</a>

8.3 Formal Definition

The described XML representation model consisting of tokens, token groups, labels and virtual values can be formalized as follows[24]:

1. There exists an input XML Mixed Content fragment `X` consisting of a sequence of `StartTags`, `EndTags`, `EmptyTags`, `NotTags` and `Attributes`:

   ```
   X := {StartTag,EndTag,EmptyTag,Attribute,NotTag}
   ```

2. The XML Mixed Content fragment `X` is interpreted as an ordered sequence of token groups. Each token group consists of an ordered triple: (1) token `T1`, which is either a `StartTag`, an `EndTag` or an `EmptyTag`, (2) a sequence of `Attribute` tokens `T2*` or `Namespace` tokens and (3) a token `T3` which is always a `NotTag` (the tokens may only appear in this order):

   ```
   TGrp(T1,T2*,T3) :=
         ({StartTag,EndTag,EmptyTag},{Attribute},{NotTag}),
              |T1|=1,|T2|=ℕ₀,|T3|=1
   ```

3. We further assign *Dewey IDs* to each token group, where `T1` is the owner of a primary label $\texttt{Lab}_{\texttt{T1}}$, and `T2` and `T3` hold labels $\texttt{Lab}_{\texttt{T2}}$ and $\texttt{Lab}_{\texttt{T3}}$ derived from the primary label. $\texttt{Lab}_{\texttt{T1}}$ is an arbitrary *Dewey ID*. $\texttt{Lab}_{\texttt{T2}}$ consists of $\texttt{Lab}_{\texttt{T1}}$ plus a negative positional number `-(pos(T2))` for each `Attribute` or `Namespace` token `T2*`. For $\texttt{Lab}_{\texttt{T3}}$ a distinction of cases is needed: (1) if `T1` is the `EmptyTag` or the `EndTag`, $\texttt{Lab}_{\texttt{T3}}$ is identical to $\texttt{Lab}_{\texttt{T1}}$; (2) if `T1` is the `StartTag`, $\texttt{Lab}_{\texttt{T3}}$ is $\texttt{Lab}_{\texttt{T1}}$ plus `0`.

   ```
   Lab(T1,T2*,T3), where
         Lab_T2 := (Lab_T1,-(pos(T2)))
         Lab_T3 := (T1=EmptyTag ∨ T1=EndTag)→(Lab_T1) ∨
                        (T1=StartTag)→(Lab_T1,0)
   ```

4. To maintain the relation between labeled token groups `TGrp` and the original input XML string `X`, a location construct `Loc` is built, which is defined as a triple containing the offset `off` of `T1` relative to the start of the original XML string `X`, the length of token `T1` $\texttt{len}_{\texttt{T1}}$ and the length of the whole token group $\texttt{len}_{\texttt{TGrp}}$:

   ```
   Loc(off,len_T1,len_TGrp)
   ```

[24] The formalization examines five tokens only, start-tags, empty-tags, end-tags, not-tags and attributes, although the model works with all other primary tokens and secondary tokens described above as well.

5. All information related to one token group can be organized in a table row `R`, where each row consists of a token group descriptor `TGrp(T1,T2*,T3)`, a label descriptor `Lab(T1,T2*,T3)` and a location `Loc(off,`len_{T1}`,`len_{TGrp}`)`.

   ```
   R := {{TGrp(T1,T2*,T3)},{Lab(T1,T2*,T3)},
               {Loc(off,lenT1,lenTGrp)}}
   ```

8.4 XML Update Operations

Now all necessary token information is available to describe the XML update language useful for the JavaScript framework. The update language is optimized for cursor-based manipulation of XML tokens in an XML Mixed Content area.

`replaceNotTag(instance,token,string)`

Replaces a not-tag token `token` with another not-tag token `string`.

`replaceTagName(instance,token,localName,namespaceUri)`

Renames a tag by overloading a name and optionally a namespace URI.

`replaceAttrValue(instance,token,value)`

Updates the value `value` of an attribute token `token`.

`insertNotTag(instance,target,offset,string)`

Inserts a not-tag token `string` into another not-tag token at an offset.

`insertEmptyTag(instance,target,offset,localName,namespaceUri)`

Inserts a new empty tag into a not-tag token `target` at an offset relative to the not-tag, by overloading a name and optionally a namespace URI.

`insertStartEndTag(instance,target1,target2,offset1,offset2,localName,namespaceUri)`

Inserts a new start-tag and a new end-tag, each into a not-tag. If `target1` and `target2` are different tokens, this corresponds to the action of wrapping a piece of XML selected with the cursor with new markup. `target1` and `target2` may also be the same not-tag token. If `offset1` and `offset2` are different, just a piece of text is wrapped with new markup. If `target1` and `target2` as well as `offset1` and `offset2` are the same, a new `StartEndTag` is inserted at a single cursor caret position.

`insertFragment(instance,target,offset,xml)`

Inserts a new fragment `xml` into a not-tag token `target` at an offset relative to the not-tag. This is useful to allow a paste action at a single cursor caret position. If a whole cursor selection shall be replaced, `removeFragment` or `removeMixed` have to be invoked first before inserting the fragment.

`insertMixed(instance,target,offset,xml)`

Same as `insertFragment`, but for a mixed token.

`insertAttribute(instance,target)`

Inserts a new attribute into a start-tag or an empty-tag token `target`.

`removeNotTag(instance,target,offset,length)`

Removes characters from a not-tag token.

`removeEmptyTag(instance,token)`

Removes an empty tag.

`removeStartEndTag(instance,token1,token2)`

Removes a start-tag and an end-tag but keeps the contents in between. This corresponds to the procedure of flattening a piece of XML Mixed Content. The operation becomes relevant if the cursor is positioned directly after an end-tag or a start-tag and the user presses the backspace key. In this situation, not only the tag directly removed by the cursor deletion but also its corresponding tag has to be removed to guarantee well-formedness.

`removeFragment(instance,token)`

Removes a fragment token, which is selected by the cursor.

`removeMixed(instance,token)`

Removes a mixed token.

`removeAttribute(instance,token)`

Removes an attribute token.

`swapFragment(instance,token1,token2)`

Swaps two tokens of the same (generic) type, for example two attribute tokens, two tag tokens or an `StartEmptyTag` and a `EmptyTag`.

9. XML Processing in JavaScript

9.1 Architecture Overview

The JavaScript framework described here is a purely client-side library and needs no special installation. The source code can be downloaded from the software repository[25] and can be deployed into an arbitrary web server directory. A start page *index.html* containing basic examples and information can be found in the topmost directory.

Three external Open Source JavaScript libraries are shipped with the framework, the *Google Closure Library* [Bol10], *jssaxparser*[26] and *CodeMirror*[27]. The Closure Library is chosen for two reasons mainly. It encapsulates cross-browser compatibility issues, for example, in the area of event handling. And it offers an object-oriented like inheritance concept to make the model-view-controller system conveniently extensible. The jssaxparser library is an Open Source SAX2 parser written in

[25] https://github.com/xrxplusplus

[26] https://code.google.com/p/jssaxparser/

[27] http://codemirror.net/

JavaScript, which is used to check the well-formedness of all XML instances loaded from outside during the initialization phase. CodeMirror provides the technical basis for the WYSIWYM control.

In this section, the state of development is described. I will put forward which parts of the use case described above could already be realized, which are pending for technical reasons and which are subject of future work because of time constraints only. Thereby, I will especially reveal the performance characteristics of the framework in combination with large documents.

All in all, the functionality of the XML model-view-controller system can be divided in seven consecutive phases, which are outlined here and described in detail afterwards:

Phase 0. If an XML editor application is implemented by means of the available XML-based user interface description language, the XML markup is transformed into HTML5 with the help of an XSLT script.

Phase 1. All XML instances are loaded into the editor by means of Ajax requests.

Phase 2. As soon as all XML instances are available, they are parsed, validated and normalized.

Phase 3. A binary index is created for one instance after the other.

Phase 4. The data bindings are calculated, respectively, the XPath expressions are evaluated against the binary index built in phase 3.

Phase 5. The binary indexes are reduced to the node-sets calculated in phase 4 (lazy encoding).

Phase 6. After stage 5, the model of the model-view-controller system is ready and the MVC system initializes the view and hereby all user interface controls.

Phase 7a. After stage 6, all static information is computed and the user can move the cursor around in the view, which comes along with XML piloting on the model side.

Phase 7b. After stage 6, the user can also manipulate data. The controller synchronizes the visual XML data in the view and the native XML data on the model-side. Phase 7b is a cyclic controlling phase. Depending on the impact of individual update operations, phase 7b can radially reference back to phase 4, phase 5 and phase 6.

9.2 User Interface Description Language

The topmost level of the JavaScript framework is a user interface description language. All higher level features such as XML instances, XML WYSIWYM controls or data binding definitions can be integrated in a browser application by either standard HTML5 markup plus attribute level definitions or alternatively by an XML UI markup language extending the HTML5 markup. As outlined in the definition in chapter 5 and in the section about XForms and the XRX architecture (chapter 2.2), a native XML editor is not only designed to *process* XML data, but it itself follows an XML-oriented *declarative programming style*.

Example 14a

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="./jssaxparser.compressed.js"></script>
    <script src="./codemirror.compressed.js"></script>
    <script src="./xrx.compressed.js"></script>
    <style>
      .xrx-instance, .xrx-bind {
        display:none;
      };
    </style>
  </head>
  <body>
    <div id="i1" class="xrx-instance" data-xrx-src="data.xml.txt"></div>
    <div id="b1" class="xrx-bind" data-xrx-ref="xpp:instance('i1')/data"></div>
    <div class="xrx-wysiwym" data-xrx-bind="b1"></div>
    <script>xrx.install()</script>
  </body>
</html>
```

Example 14b

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="./xrx2html.xsl" type="text/xsl"?>
<html>
  <head>
  </head>
  <body>
    <xrx:instance id="i1" src="data.xml.txt"/>
    <xrx:bind id="b1" ref="xrx:instance('i1')/data"/>
    <xrx:wysiwym bind="b1"/>
  </body>
</html>
```

Example 14a and example 14b are minimalist, simplified XML editor applications to illustrate the two alternatives. Example 14a is a pure HTML5 document including an instance definition, a data binding definition and a WYSIWYM control represented by the HTML5 DIVs with the class names `xrx-instance`, `xrx-bind` and `xrx-wysiwym`. Since `xrx-instance` and `xrx-bind` are components belonging to the model of the MVC system—which means they are providing just data logic—they are made invisible with a CSS rule `display:none`. In the header, the JavaScript files needed are included: the jssaxparser library, the CodeMirror library and the JavaScript framework, identifiable with the `xrx` prefix. Right below, the XML editor application is initialized with a `xrx.install()` JavaScript call. For `xrx-instance` and for `xrx-bind`, it is important to have a unique identifier attached (`i1` and `b1`), since these components can be referenced by other components of the application. The `xrx-bind`, for example, references the data hold by `xrx-instance` by means of a custom XPath function `xrx:instance('i1')/data` defined in the HTML5 `data-xrx-ref` attribute. The WYSIWYM control `xrx-wysiwym` in turn accesses the data held by `xrx-bind` with the id specified in its `data-xrx-bind` attribute.

Example 14b shows exactly the same application based on the XML UI markup language. Formalisms such as the inclusion of JavaScript libraries as well as the initialization call are outsourced to the XSLT script *xrx2html.xsl*. The XSLT script transforms the XML UI language into exactly the HTML5 document introduced as example 14a. All in all, the XML-based source code is more compact and could be validated against an XML Schema. XForms authors might feel more confident with the XML UI language approach, HTML authors might prefer the HTML5 alternative [Lain12].

On the model side, the UI description language closely follows the XForms standard, on the view-side, it also offers custom controls, e.g. `xrx:wysiwym`, with markup tags different from the XForms markup language. It is not clear at the moment if it makes sense to design the user interface description language as a strict extension of the XForms standard or as just another custom UI description language. Some criticisms about the usefulness of XForms as a general UI description language beyond forms-oriented UI design do not support a strict XForms approach [Pohj10] [Fons07]. Within the SCA community and also in the TEI XML community, graphical user interface controls play an important role as well, e.g. controls for image annotation or for text-image-linking [Col11] [Dipp11] [Kayl09]. At the moment it seems to me that extending the XForms markup language in direction of a common UI description language including graphical controls overexpands the original form-oriented design of the XForms markup language. The model related XForms markup to describe data and application logic, however, seems useful for all kinds of XML intensive browser applications.

9.3 Loading Stringified XML Instances

The JavaScript framework for visual and native XML editors provides an alternative XML representation model and an alternative XML processing system replacing the browser DOM. To establish such an alternative, the characteristic of a browser's *XMLHttpRequest* object[28] is utilized, similar as described in [Mil11]. XMLHttpRequest is the standard interface to exchange data between a server and a browser client within Ajax applications and is supported by various browsers since 1999 [Mil11]. Slight browser-specific variances exist, which are encapsulated by the Closure Library in the framework. Unlike the prefix *XML*HttpRequest suggests, the API is not only used to request XML data within Ajax applications, but for all kinds of data such as JSON and especially also for plain text documents.

The process of loading an XML document into a browser normally works as follows: once an XML request has been made, two objects are created by XMLHttpRequest, (1) a `responseText` object of type `DOMString`, which holds a

[28] http://www.w3.org/TR/XMLHttpRequest/

decoded string representation of the XML document loaded, and (2) a `responseXML` object, which is of type DOM `Document`. The `responseText` object is an intermediary read-only object used by a browser to parse and validate the requested XML or (X)HTML data and to further transform them into an according DOM representation. If other than XML data are requested, maybe JSON data, the contents of the `responseText` object are not further processed. They can be used by Ajax applications though, e.g. to parse and create JSON objects. The behavior of the XMLHttpRequest interface is influenced by the media-type specified in the HTTP request header. Only if the media-type is `application/xml`, `text/xml` or another `+/xml` type, the browser starts DOM parsing, in all other cases XMLHttpRequest stays passive.

The approach used by the JavaScript framework to go round the browser DOM is thus straightforward. The only precondition to replace the browser DOM with an alternative memory-optimized XML representation model—which is at the same time the only server-side dependency of the JavaScript framework, if you will—is that XML instances have to be served with an actually wrong media-type, namely as `text/plain` documents. It is to remark that XML documents need no server-side transformation, but just a different media-type when sent. The media-type `text/plain` makes the browser believe that a plain text document is provided by the server, and at the same time, the media-type avoids that the browser starts DOM parsing. Instead, the `responseText` object can be directly used by the JavaScript framework for streaming XML processing or to create binary XML indexes. For test purposes, it is already enough to serve XML documents with the file extension `*.xml.txt` for example. The file extension `*.txt` is sufficient to make the browser believe it is a pure text document. Serving XML instances with media-type `application/xml` to the JavaScript framework or with the file extension `*.xml` works as well, since the `responseText` object is created in either case by XMLHttpRequest as an intermediary object. The memory-optimizations of the JavaScript framework would not be effective then, though.

Plain-text XML instances are sent back to a server as ordinary XML documents with the right media-type `application/xml.` There is nothing special to respect for incoming XML documents from a server-side application's point of view. Testing the procedure to send XML strings with media-type `application/xml` with various browsers shows that the value of the `responseXML` object is still null after the Ajax call is complete. The memory-optimized XML model of the JavaScript framework is intact for the send situation too, since browsers do not create a DOM object before sending an XML string with media-type `application/xml`.

There is one little issue concerning XML entities for the XML string approach, which needs attention: interactive user interface controls that aim to make XML entity encoded characters editable based on the JavaScript framework cannot directly rely on the browser's standard behavior. Browsers normally decode HTML and UTF-8 entities implicitly into an according human-readable representation, but only in the case of XML documents and not in the case of plain-text documents. Entities thus appear in their raw form in an editing field at first, and the user interface controls themselves are responsible for providing a readable and editable representation.

9.4 XML Parsing, Reading, Streaming, Traversing and Piloting

Once the XML instances are loaded, the model-view-controller continues to initialize the XML editor application by parsing and validating all XML instances. For this, the jssaxparser library is used. The guarantee to have well-formed XML instances available is crucial for the XML reading, streaming, traversing and piloting classes, since they are designed to work in both directions, in forward as well as in backward direction. During the parsing phase, the XML instances are also normalized, which means that ignorable whitespaces and indentations are removed. If a start-tag, for example, is served in the following whitespace afflicted although well-formed way, `<a attr =  "value"   />`, it appears in compact form in the XML stream, `<a attr="value"/>`, after normalization. Streaming works faster if it is certain that one can expect exactly one space between a tag-name token and an attribute token, for example, since this reduces the number of code instructions during XML streaming.

Example 15: Public Interface of the XML Streaming Class

```
// primary tokens
xrx.stream.prototype.rowStartTag = function(offset, length1, length2) {};
xrx.stream.prototype.rowEndTag = function(offset, length1, length2) {};
xrx.stream.prototype.rowEmptyTag = function(offset, length1, length2) {};
xrx.stream.prototype.rowComment = function(offset, length1, length2) {};
xrx.stream.prototype.rowPI = function(offset, length1, length2) {};
// secondary tokens
xrx.stream.prototype.eventTagName = function(offset, length1, length2) {};
xrx.stream.prototype.eventAttribute = function(offset, length1, length2) {};
xrx.stream.prototype.eventAttrName = function(offset, length1, length2) {};
xrx.stream.prototype.eventAttrValue = function(offset, length1, length2) {};
xrx.stream.prototype.eventNamespace = function(offset, length1, length2) {};
xrx.stream.prototype.eventNsPrefix = function(offset, length1, length2) {};
xrx.stream.prototype.eventNsUri = function(offset, length1, length2) {};
```

All XML classes described in this section are event-based XML APIs, which share some characteristics with SAX and StAX, but also are different in some respects. The XML streaming classes follow the StAX API in the ability to skip XML events. As a default, only events for primary tokens (see chapter 8.1.1) are thrown, respectively, an event for each XML token `row` is thrown in any case. Secondary tokens (see chapter 8.1.2) are ignored by default and can be switched on and off by other classes as desired. For example, the XML pilot class, which handles cursor moves, normally ignores secondary tokens, whilst the index builder needs at least namespace information and thus switches several secondary tokens to on. In contrast to StAX, the streaming classes can skip

XML tokens not only on the event level but also on the syntactical level. If secondary tokens are turned off, the streaming class runs over as many syntactical details as possible to reduce the number of code instructions again. In contrast to SAX and StAX, the streaming classes work in forward as well as in backward direction. Reading XML instances from end to start is possible, as soon as well-formedness is checked and also as soon as all namespace definitions are known. Currently, namespace definitions are analyzed during the index building phase, where each XML instance is streamed from start to end once, as a whole. The XML streaming classes in addition follow StAX in being able to stop streaming at any token of an XML instance and also in being able to continue or to start parsing somewhere in the middle. To keep the streaming classes simple, streaming starts are possible at the beginning of a token `row` only, but not inside a tag token, at an attribute token or inside a text token. This requires the re-streaming of XML tokens in some situations, but all in all keeps the streaming classes simple.

While the streaming interface (example 15) can be used to analyze XML tokens, the XML traverse class in addition calculates the structural characteristics of an XML tree. It is the central class, which implements the XML labeling scheme defined in chapter 8.3. It supports the same XML event interface like the streaming class (example 15) with the difference of an additional Dewey ID label provided as the first event function parameter. With the help of the XML traverse class, it is especially possible to evaluate all kinds of XPath axes. Streaming evaluation of forward axes always works straightforwardly. In contrast to other streaming XPath processors that find sophisticated solutions to evaluate backward axes, the JavaScript framework offers naive backward axis evaluation by means of simply traversing backward. The only axis requiring a special treatment is the preceding axis. For example, if a streaming class has to reach the tag token with label `1.3.5` and starts its evaluation at tag `1.3.12.7`—which means a backward evaluation of two tokens that lie in the preceding axis—there is an indirection needed: the joint parent tag with label `1.3` has to be reached in backward direction first and only after that can tag `1.3.12.7` be accessed in forward direction. So, to allow a naive and purely streaming-based evaluation of the preceding axis without node buffering required, a zigzag course approach is introduced. Performance of preceding axis evaluation is not a critical issue for an XML editor application, since the preceding axis is normally not used in the area of global XPath data binding expressions, but only for local XML piloting.

Table 4a: XML Reading

	Chrm	FF	IE
Hamlet forward	1ms	6ms	7ms
Hamlet backward	1ms	6ms	7ms
CSGIII forward	9ms	62ms	93ms
CSGIII backward	8ms	61ms	102ms
6MB forward	18ms	120ms	182ms
6MB backward	16ms	116ms	181ms

Table 4b: XML Traversing (secondary tokens off)

	Chrm	FF	IE
Hamlet forward	7ms	12ms	23ms
Hamlet backward	8ms	15ms	28ms
CSGIII forward	47ms	126ms	197ms
CSGIII backward	59ms	139ms	225ms
6MB forward	90ms	220ms	364ms
6MB backward	101ms	270ms	388ms

Table 4c: XML Traversing (secondary tokens on)

	Chrm	FF	IE
Hamlet forward	30ms	39ms	95ms
Hamlet backward	29ms	38ms	95ms
CSGIII forward	241ms	328ms	907ms
CSGIII backward	240ms	366	920ms
6MB forward	427ms	761ms	1453ms
6MB backward	429ms	791ms	1478ms

Table 4d: XML Parsing (forward only)

	Chrm	FF	IE
Hamlet	86ms	114ms	244ms
CSGIII	888ms	1310ms	2741ms
6MB	1487ms	1911ms	5152ms

Table 4a to table 4d inform about the performance characteristics of the JavaScript XML processing system. The benchmark tests were performed on a customary 32 bit operating system equipped with 2 CPUs, 3GHz and 2GB RAM. Three browsers served as testing environments: *Google Chrome* version 32 (Chrm), *Mozilla Firefox* version 27 (FF) and *Internet Explorer* version 11 (IE). The three test files introduced in chapter 6.3 were processed: *hamlet.xml* (300 kilobytes), *CSGIII.xml* (3000 kilobytes) and *6MB.xml* (6000 kilobytes). All classes were tested in forward as well as in backward direction, except the SAX parser which works in forward direction only. Each test was executed three times in succession. The measured result was accepted only if the standard deviation was less than 10 percent, otherwise the test was retried. The XML files were loaded by means of the XMLHttpRequest interface as described in the last section. They were also parsed and normalized beforehand to simulate the XML processing classes under as most real as possible conditions. It is to be remarked that the *6MB.xml* test file does not contain any line breaks after normalization any more. It is a 6 megabytes long string in the end, consisting of about 5.7 million characters. The benchmark tests are available as part of the source code and can be reproduced if desired.

The XML reading class is the most low level class of the XML processing system. The benchmark table 4a simply informs about the time span needed to run through a large XML string with a browser character by character without doing anything. Google Chrome is outstanding in reading the 6 megabytes large string in only 16 milliseconds. This means to run through a text document of about 1000 printed pages in only 16 milliseconds. Firefox and Internet Explorer still perform well. Firefox works about seven times slower and Internet Explorer about ten times slower than Chrome. All in all, one can say that a browser is a viable environment for reading large text documents. Such large strings without line breaks would be processed with some kind of buffered string reading in a Java or C-based program. The JavaScript string implementations provided by the tested browsers behave convenient. Buffered reading is obviously internally handled by the JavaScript environments.

Table 4b and table 4c inform about the performance of the XML traversing class, the class which provides token information and additionally Dewey ID labels. Chrome loses its clear advance here. Whilst traversing in Chrome is more than five times slower than reading, reading and traversing with Firefox and Internet Explorer differs only by factor two to three. It is noticeable that traversing the primary tokens of an XML instance only (table 4b) is about four to five times faster than traversing the secondary tokens as well (table 4c) for all browsers. To traverse an XML instance in any detail by visiting each token of an XML instance without exception like a SAX parser, in turn, is still about three times faster than parsing an XML document with an ordinary SAX parser (table 4d). The results demonstrate that the introduction of an extra streaming and traversing class different from the XML parser is a worthwhile undertaking in any case. The table also illustrates that the XML pilot, which internally makes use of the traversing class (with primary tokens on only), can reach each node of a 6 megabytes large XML instance in only 388 milliseconds, even with the slowest of all browsers and even if it has to pilot from the very start to the very end of the document, which is a seldom case. The performance of forward and backward processing generally does not differ for all browsers, files and classes.

All in all, it turns out that the string processing capabilities of browsers hand in hand with the XML traversing approach of the JavaScript framework create favorable terms for streaming XML processing. Cursor-based XML manipulation and incremental XML synchronization in a model-view-controller driven application thus are tasks that oblige the JavaScript environment. The most critical point at the moment is the time needed for XML parsing when the XML editor and the XML instances are loaded the first time. The more than five seconds needed by Internet Explorer for a file of 6 megabytes are not an acceptable situation. It could not yet be figured out where exactly the time is lost and if parsing performance can be improved. It would be an interesting experiment to cross-compile one of the mature Java or C-based SAX parsers or StAX parsers into JavaScript to see if the parsing performance is a problem for JavaScript in general. Due to the restrictions of the XML parsing stage with Internet Explorer, it is not recommended to use the JavaScript framework for documents larger than one megabyte at the moment.

9.5 Building Binary XML Indexes

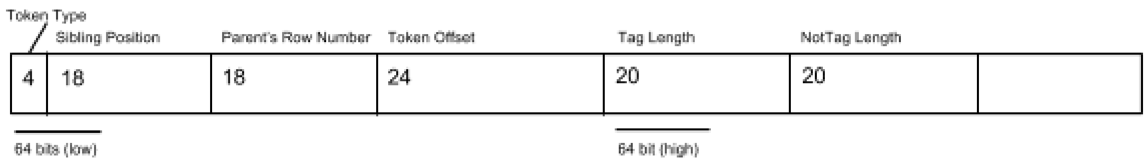
In this section, a binary version of the XML representation model defined in chapter 8 is proposed. Compact in this context means to encode all tokens and token groups of a XML instance with as little information as possible, with a consistent binary token descriptor, and with binary blocks of equal size.

Table 5: Compact Encoding of Dewey ID Labels

Row Number	Dewey ID	Compact Encoding ([Position] [Parent])
0	1	[1] []
1	1.1	[1] [0]
2	1.1	[1] [0]
3	1.2	[2] [0]
4	1.3	[3] [0]
5	1.3.1	[1] [4]
6	1.3.1	[1] [4]
7	1.3	[3] [0]
8	1	[1] []

Dewey ID labels differ in size depending on the nesting depth of a specific token at first. Table 5 shows the approach implemented by the JavaScript framework which encodes Dewey IDs evenly in a double-digit form. A Dewey ID label can be interpreted by (1) the positional number of a token in a list of sibling tokens represented by the last label item and by (2) the label of the parent. The label of the parent token is already encoded somewhere in the list of labels, however. Thus it is not necessary to encode the parent label again, but it is sufficient to reference the parent label’s row number in the list of labeled token groups. A Dewey ID label (column 2 of table 5) in a consecutive list of Dewey ID labels thus can be compactly encoded (column 3) by a double-digit number array consisting of the sibling position and the parent’s row number (column 1).

Figure 3: Binary Token Descriptor



The token descriptor currently implemented by the JavaScript framework uses a 128 bit binary mask consisting of 64 low bits and 64 high bits (figure 3) to encode one token group. 2^4 bits are used for the token type and 2^{18} bits for the sibling position of the token label and the row number of the token's parent label at any one time. The token offset gets 2^{24} bits and the tag length and the length of the token group 2^{20} bits respectively. The last 24 bits remaining are left open to reference token related XML Schema information, once an XML Schema implementation is available.

With 2^{24} bits, it is theoretically possible to encode documents up to about 16.7 million characters, which corresponds to a document size of about 18 megabytes. This would theoretically mean that, in an extreme case, a document could contain a list of 2^{22} (about 4.2 millions) sibling elements, since empty tag tokens can be only 4 characters long. The maximum number of sibling elements is limited to 2^{18} (262 thousand), however, in the token descriptor, since this seems already a generous number of sibling tokens in a real-world XML document of 18 megabytes size. Also it seems sufficient for real-world documents to limit the length of tag tokens and not-tag tokens to 2^{20} (about 1 million) characters. It is not clear at the moment whether the token descriptor will stay exactly this way. In fact, it is implemented in the form of an easy configurable binary mask object to experiment with different bit sizes. Future versions of the JavaScript framework may use variants depending on the XML documents to be processed.

Currently, there are two alternatives to work with binary data in a web browser: a very recent browser technology, so-called typed arrays that offer native binary data support, and the older provisional approach that uses ordinary integer or string objects as a medium for binary encodings. Typed arrays originate in the WebGL specification in conjunction with the HTML5 movement. Although there exist just some drafts at the moment and no final recommendation[29], most modern browsers already have support for native binary data. At least the *ArrayBuffer* object and the *ArrayBufferView* object are available for the most part. Since the usage of the modern JavaScript binary objects would exclude too many browser clients and thus too many users at the moment, I decided to start with a conservative binary index implementation. The JavaScript framework currently uses a standard JavaScript *Array* object to collect binary encoding blocks. The binary encoding blocks themselves are realized with the help of the *goog.math.Long* object provided by the Closure Library, which simulates a 64-bit two's-complement integer value and internally relies on standard JavaScript integers[30]. Two

[29] https://www.khronos.org/registry/typedarray/specs/latest/

[30] http://docs.closure-library.googlecode.com/git/class_goog_math_Long.html

goog.math.Long objects for each token group are sufficient to realize the token descriptor above. The approach works for older browsers up to Internet Explorer 7. To utilize the advantages of the native binary data implementations of today's browsers, a browser chain that chooses between native and provisional binary support depending on the client, has to be implemented. The browser chain including modern native binary data support could not be implemented yet because of time constraints.

9.6 XPath Support

The XPath implementation integrated into the JavaScript framework for visual and native XML editors is a further development of the *Wicked Good XPath* implementation[31]. The arbitrarily streaming or binary-based working XPath processor was gained by at first outsourcing all node-specific functions found in the Wicked Good XPath processor into a single class `xrx.node`. The result is the function-set shown in table 6. It summarizes all node-specific functions that are necessary to make the XPath processor work. In reformulating the `xrx.node` class into an abstract class, the processor was made node agnostic. The originally browser DOM-based node evaluation functions were removed and replaced by two custom node implementations, a binary node implementation `xrx.nodeB` and a streaming node implementation `xrx.nodeS`, which both implement the `xrx.node` interface. These node classes internally rely on the token-based XML model described in chapter 8. Since the Wicked Good XPath library does not support XML Namespaces, the XPath `NameTest` class was revised and extended with namespace related functionality.

Table 6: Interface for a Node Agnostic XPath Processor

```
// value conversion functions
xrx.node.prototype.getValueAsString = goog.abstractMethod;
xrx.node.prototype.getValueAsNumber = goog.abstractMethod;
xrx.node.prototype.getValueAsBool = goog.abstractMethod;
// node identity and document order functions
xrx.node.prototype.isSameAs = goog.abstractMethod;
xrx.node.prototype.isBefore = goog.abstractMethod;
xrx.node.prototype.isAfter = goog.abstractMethod;
// axes test functions
xrx.node.prototype.isAncestorOf = goog.abstractMethod;
xrx.node.prototype.isAttributeOf = goog.abstractMethod;
xrx.node.prototype.isChildOf = goog.abstractMethod;
xrx.node.prototype.isDescendantOf = goog.abstractMethod;
xrx.node.prototype.isFollowingOf = goog.abstractMethod;
xrx.node.prototype.isFollowingSiblingOf = goog.abstractMethod;
xrx.node.prototype.isParentOf = goog.abstractMethod;
xrx.node.prototype.isPrecedingOf = goog.abstractMethod;
xrx.node.prototype.isPrecedingSiblingOf = goog.abstractMethod;
```

[31] https://code.google.com/p/wicked-good-xpath/

```
// name functions
xrx.node.prototype.getName = goog.abstractMethod;
xrx.node.prototype.getNamespaceUri = goog.abstractMethod;
xrx.node.getNameLocal = goog.abstractMethod;
xrx.node.getNamePrefix  = goog.abstractMethod;
xrx.node.getNameExpanded  = goog.abstractMethod;
// content functions
xrx.node.prototype.getStringValue = goog.abstractMethod;
xrx.node.prototype.getXml = goog.abstractMethod;
// axes evaluation functions
xrx.node.prototype.getNodeAncestor = goog.abstractMethod;
xrx.node.prototype.getNodeAttribute = goog.abstractMethod;
xrx.node.prototype.getNodeChild = goog.abstractMethod;
xrx.node.prototype.getNodeDescendant = goog.abstractMethod;
xrx.node.prototype.getNodeFollowing = goog.abstractMethod;
xrx.node.prototype.getNodeFollowingSibling = goog.abstractMethod;
xrx.node.prototype.getNodeParent = goog.abstractMethod;
xrx.node.prototype.getNodePreceding = goog.abstractMethod;
xrx.node.prototype.getNodePrecedingSibling = goog.abstractMethod;
```

The largest part of the Wicked Good XPath processor could be taken unaltered, since XPath implementations are largely node-independent in general: (a) the parsing of the XPath expressions, its validation and conversion into an automaton, (b) the overall evaluation plan that finds a strategy to evaluate the single constituents of the automaton and (c) the nodeset class which orders the results and omits duplicates from the result set. Besides that, there are a lot of XPath functions and operators defined in the XPath specification that are node-independent anyway, for example, string functions or logical operators.

9.7 Risk-free XML WYSIWYM Authoring

In this section, I will present the WYSIWYM control for risk-free XML Mixed Content authoring. It is realized by means of a visual HTML representation construct with XML token labels attached, which is made interactively editable by means of a code editor software.

As already outlined above, browsers completely eliminate all HTML tags in a *contentEditable* field and only present (formatted) text to the user, which is convenient for layout-oriented text annotation but not necessarily for deep nested XML markup annotation, as summarized in the caret cursor position argument (CCPA). As a consequence namely, only text edges are available for cursor caret-oriented authoring, but not the markup edges. Desktop style XML WYSIWYM editors, in contrast, preserve at least a placeholder string for each markup tag or even provide advanced visualizations for semantic tags [Bin13]. Through this, all markup authoring becomes cursor-oriented. The JavaScript framework described here follows the placeholder approach of desktop style editors, gives up the *contentEditable* feature and implements its own web WYSIWYM control based on a code editor software.

Table 7: The Three Representational Models of a WYSIWYM Control

Raw XML Representation	**HTML Representation**	**Visual Representation**
<a>Text <b/>text<c>text</c></a>	<span class='start-tag'>↦</span> <span class='text'>Text </span> <span class='empty-tag'>↥</span> <span class='text'>text</span> <span class='start-tag'>↦<span> <span class='text'>text</span> <span class='end-tag'>↤</span> <span class='end-tag'>↤</span>	↦Text ↥text↦text↤↤

Table 7 shows the three representational models of a WYSIWYM control. In the first column, the raw XML Mixed Content fragment is shown. The second column shows the according HTML representation consisting of HTML SPANs mainly. HTML SPANs are not editable normally, but can be changed so by means of code editor software. Placeholder strings (↦, ↤ and ↥) are used to mark the edges of start-tags, end-tags and empty-tags. The third column shows the visual XML MIxed Content fragment, as it would appear on the screen to the user. Using placeholders for semantic tags makes the markup edges accessible for cursor manipulation. Additionally, the HTML SPANs surrounding the placeholder strings can be used for individual CSS styling or to construct advanced JavaScript or SVG-based visualizations. The JavaScript framework at the moment realizes at least highlighting for the tags and the text which is currently near the cursor.

Table 8: Applying the Token-Based XML Labeling Scheme to the XML WYSIWYM Control

1	`<a`**`{1}`**`>1`**`{1.0}`**	`<span`**`{1}`**`>↦</span><span`**`{1.0}`**`>1</span>`
2	`<b attr='value'`**`{1.1}`**`/>2`**`{1.1}`**	`<span`**`{1.1}`**`>↥</span><span`**`{1.1}`**`>2</span>`
3	`<c`**`{1.2}`**`>3`**`{1.2.0}`**	`<span`**`{1.2}`**`>↦<span><span`**`{1.2.0}`**`>3</span>`
4	`</c`**`{1.2}`**`>4`**`{1.2}`**	`<span`**`{1.2}`**`>↤</span><span`**`{1.2}`**`>4</span>`
5	`</a`**`{1}`**`>`	`<span`**`{1}`**`>↤</span>`

Table 8 shows how a raw XML Mixed Content fragment and its visual HTML representation are interconnected by means of the token-based labeling scheme formalized in chapter 8.3. HTML SPANs get the same labels attached as their according XML tokens. It is to remark, that the HTML representation contains primary token information only. If tag-name information or attribute information is required during authoring, this can be derived dynamically from the raw XML instance. All in all, the HTML representation is free of any semantic information. In contrast to the SCA tools and transcription tools introduced above, the HTML model contains information about XML tokens and their hierarchical structure only. It is a generic model useful for XML documents in general and is not designed to represent the semantics of a specific XML standard. The raw XML document stays the primary representation for the whole authoring cycle and the HTML construct is only a thin layer for visualization.

The labels in the HTML construct can be, above that, used to guarantee well-formedness. A XML Mixed Content fragment becomes invalid, for example, if a start-tag is removed but not its according end-tag. With the help of the code editor software, the WYSIWYM control keeps track that the corresponding end-tag of a removed start-tag is always deleted at the same time. Also cursor selections can be handled in this way: a selection is valid only if the label of the left cursor edge and the label of the right cursor edge are child tokens of the same parent token. If this is not the case, authoring would lead to invalid markup. Thus, authoring is only allowed by the code editor software if the cursor selection is valid.

9.8 The Update-Recalculation Cycle

The update-recalculation cycle of the XML model-view-controller system covers five areas: (1) updating raw XML instances, (2) updating namespace tables, (3) relabeling and rebuilding binary encodings, (4) re-evaluating XPath expressions and (5) refreshing user interface controls. The aim is to find an individual recalculation strategy for all five areas and for each of the update operation defined in chapter 8.4 in order to minimize recalculation costs. Thus, all user interface controls that take part in the model-view-controller system never update a raw XML instance, a namespace table or a binary index directly, but deliver the individual handling of update operations to the controller class `xrx.controller`. The controller class decides how to deal with an update operation in an optimal way. In this section, the basics of the update-recalculate controlling strategy are outlined. All in all, this part of the JavaScript framework is the currently most least developed part.

(1) Updating raw XML instances. Updating raw XML instances is the most uncritical task. XML instances are updated by means of a naive string split and join mechanism:

```
var tmp = this.xml_.substr(0, offset) + update + this.xml_.substr(offset
+ length);
this.xml_ = tmp;
```

The complete raw XML string is split into two parts with the help of the JavaScript `substr` method. The update string is inserted in between the two temporary substring copies. All parts are joined and declared as the new XML string. This naive approach works even for very large strings without performance losses on all tested browsers.

(2) Namespace tables. Namespace tables can be updated straightforward as well. Only tag token removals or insertions necessitate namespace table updates. The most complex case is the `removeStartEndTag` operation. Since a start-tag and an end-tag and possibly a namespace definition are removed at once here, but without removing the inner tags between the `StartEndTag` that possibly depend on the namespace definition of the `StartEndTag`. So, the namespace definition may not be simply removed for this update operation but has to be moved somewhere. Also, removing or inserting fragment tokens or mixed tokens sometimes result in updating more than one namespace in a table. Namespace related update procedures can be quite complex, but they have no negative performance influence since namespace definitions do not appear in great quantities.

(3) Binary index. For the binary index, there exist two critical factors: (a) the number of *structural* updates versus *value* updates only and (2) the size of an index. There exist value updates that require recalculation of offset and length location encodings only, and structural updates that additionally require label recalculation in a binary XML processing system. Label updates are generally more expensive than just location updates. As described in chapter 8, the relabeling costs are already minimized by the model itself, since only tag removals and tag insertions imply a structural update. Text manipulation as well as all kinds of attribute modifications are value updates only according to the model, thanks to the empty not-tag token approach and thanks to the just virtual encoding values for attribute and not-tag tokens.

Each structural and value update still necessitates the update of offset and length information in the binary index though. The more location encodings exist, the more encodings have to be recalculated. The JavaScript framework finds some first strategies to reduce the costs, enabled by the hybrid XML processing system: firstly, only primary tokens are indexed by the JavaScript framework. To process secondary tokens like tag-names or attribute-values, the hybrid XML processing system switches from binary to streaming evaluation. So, in contrast to binary-based XML processors such as native XML databases, the JavaScript processor currently does not create long-living encodings for tag-names, attributes or other secondary tokens. This reduces the number of encodings considerably but in turn makes XPath evaluation slower. Finding an adequate balance between the number of long-living encodings and XPath performance is subject of current investigations.

Secondly, there is support for lazy encoding in progress: all tokens that are not matched by the XPath binding expressions during the data binding phase can be generally removed from the index and thus reduce its size. As described earlier, it is a typical situation for an XML editing application that the outer nodes of an XML document are statically bound to UI controls whilst the inner Mixed Content areas are dynamic and frequently updated. A favorable characteristic of the Dewey ID labeling scheme in turn is that the labels of whole subtrees of a labeled XML instance can be removed from the model and inserted again later without affecting other labels or breaking the model as such. Since there is a streaming XML API available in the hybrid system in any case, all kinds of XML operations are still possible on subtrees, even without binary encodings.

(4) XPath data binding expressions. Using XPath as a data binding language is an extraordinary use-case with no research available at the moment, if I see this right. Some native XML databases implement XML node path tables, but only for simple paths and not for XPath expressions [Gruen10]. Normally XPath is used for querying nodes ad hoc, but not for node addressing that shall endure. The situation is particular, since the binding expressions are static during the life cycle of the XML application and the model at the same time is dynamic.

As far as I can see now, there exist two ways to decide if a particular XPath expression is subject of recalculation. The fastest way is to decide about recalculation on the XPath expression level. The second more expensive way is to decide on the node level.

Assuming that there exists an XPath expression `/a/b` and the model is updated with an `replaceAttrValue` operation, the result set of expression `/a/b` certainly does not change. For an expression like `/a[@b='c']/d` and an update operation `removeAttribute` the decision is more difficult, but the name of the attribute token can be used as the deciding factor: if the name of the removed attribute is not equal 'b', it is sure that the result-set of `/a[@b='c']/d` does not change by the model update. If the name is 'b', an additional node-test is required.

There are other XPath-expression-update-operation combinations that generally cannot be handled on the expression level. An expression like `/a/b//c` and an update operation that removes a node 'c' from the tree is subject of recalculation only if the removed node is part of the result set of expression `/a/b//c`. It is not clear yet, however, if deciding about recalculations on the node level is perhaps sometimes more expensive than just recalculating the whole binding expression. Using XPath as a data binding language in combination with a dynamic XML model is a currently open-end question and subject of investigation.

(5) Refreshing the view. A general quality criterion for a model-view-controller is to avoid complete view refreshes. The aim is to refresh only those controls of a view that are affected by a specific data update. There exist two situations which have to be considered, XPath-related refreshes and node-related refreshes: (a) if an XPath data binding expression is recalculated and the result-set of the data binding changes consequently, also all controls that reference the binding have to be refreshed. (b) If two user interface controls are bound to the same node or if two controls are bound to nodes that lie in the ancestor-descendant axis, there is a refresh required after an update.

10. Conclusion

In this thesis, a JavaScript framework for visual and native XML editors is described, which intends to make the development of browser-based XML WYSIWYM editors as natural as in the Java or C world. Two main barriers are identified, the browser DOM and the browser built-in WYSIWYG feature *contentEditable*. The thesis investigates whether the emulative capabilities of the JavaScript environment are powerful enough to overcome the discovered browser limitations. Within the investigation, the browser DOM is replaced by a hybrid XML API that is streaming and binary compatible. Furthermore, a custom WYSIWYM control is implemented. The thesis concludes that JavaScript is a viable environment for streaming XML processing of relatively large XML documents due to its excellent string processing capabilities. Binary XML processing is possible as well with JavaScript. Since support for native binary data processing in the browser is a currently emerging technology, rapid improvements are to be expected in this area within the next years.

Implementing a WYSIWYM control in JavaScript is straightforward thanks to the groundwork done by code editor software. The WYSIWYM control integrated in the framework is simplistic at the moment and needs more sophisticated visualization techniques. The set of available built-in user interface controls in the XML MVC system should be extended as well, amongst others with form controls. Lastly, the documentation of a common MVC interface for the convenient integration of domain specific UI components, e.g. image annotation or text-image-linking components, is desirable. An integral component missing at the moment, however, is an XML Schema processor for document validation and for target markup adoption. A JavaScript XML Schema processor should work for large XML documents as well and additionally should respect the special use case of an XML editor application: cursor-oriented XML manipulation requires partial and incremental validation of XML documents, an XML Schema interface to realize context sensitive markup suggestions as well as support for customizable XML Schema error reports.

11. Acknowledgements

First of all, I would like to thank Prof. Dr. Manfred Thaller for his remarkably interesting seminars about Humanities Computer Science at the University of Cologne. They gave me the main impetus to investigate and realize an XML processing system as described here. Also, I'd like to thank Georg Vogeler, the developer of the so-called *Charters Encoding Initiative* (CEI) XML standard. In many emails and discussions, he gave me a clear picture of what is expected from a web-based XML editor from an XML Schema developer's point of view. A software tool is useless as long as nobody uses it. Thus, I'd like to thank Thomas Aigner and Karl Heinz, the leading minds of the platform *Monasterium.net*. With them, there will be the invaluable chance to develop the XML editor framework on a real web platform in cooperation with a very active and tool-excited user community. Last but not least, I would like to thank Katharina Mahler and Wibke Pfeiffer for helping me to improve the linguistic style and the layout of this thesis.

12. References

[Bin13] George Bina. 2013. *Customizing a general purpose XML editor: oXygen's authoring environment*. In: Proceedings of the International Symposium on Native XML User Interfaces. Balisage Series on Markup Technologies, vol. 11 (2013). DOI=10.4242/BalisageVol11.Bina01

[Cam13] Stephen Cameron, William David Velásquez. 2013. *A Data-Driven Approach using XForms for Building a Web Forms Generation Framework*. In: Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). DOI=10.4242/BalisageVol10.Velasquez01

[Ebn13] Daniel Ebner. 2013. *XRXEditor - ein generischer schemabasierter XForms Editor*. URL=http://www.vdu.uni-koeln.de/vdu/xrx-editor/tool-description [date accessed: 13 Dec. 2013]

[Flyn13] Peter Flynn. 2013. *Could authors really write in XML one day?*. In: Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). DOI=10.4242/BalisageVol10.Flynn02

[Khal13a] Ali Khalili, Sören Auer. 2013. *User Interfaces for Semantic Authoring of Textual Content: A Systematic Literature Review*. In: Web Semantics: Science, Services and Agents on the World Wide Web, North America (September 2013)

[Khal13b] Ali Khalili, Sören Auer. 2013. *WYSIWYM–Integrated Visualization, Exploration and Authoring of Un-structured and Semantic Content*. URL=http://www.semantic-web-journal.net/system/files/swj479.pdf [date accessed: 13 Dec. 2013]

[Maal13] Mustapha Maalej, Anne Brüggemann-Klein. 2013. *Generating Schema-Aware XML Editors in XForms*. In: Proceedings of the International Symposium on Native XML User Interfaces. Balisage Series on Markup Technologies, vol. 11 (2013). DOI=10.4242/BalisageVol11.Bruggemann-Klein01

[Oliv13] Bruno Oliveira, Vasco Santos, Orlando Belo. 2013. *Performance Analysis of Java APIs for XML Processing*. URL=http://sdiwc.net/digital-library/performance-analysis-of-java-apis-for-xml-processing [Date accessed: 13 Dec. 2013]

[Dal12] Karina van Dalen-Oskam, Joris Job van Zundert. 2012. Digital Editions with eLaborate: from practice to theory. URL=http://www.dh2012.uni-hamburg.de/conference/programme/abstracts/digital-editions-with-elaborate-from-practice-to-theory/ [Date accessed: 13 Dec. 2013]

[Den12] Greg Dennis, Joon Lee. 2012. *Wicked Good XPath: a faster JavaScript XPath library*. URL=http://google-opensource.blogspot.de/2012/09/wicked-good-xpath-faster-javascript.html [Date accessed: 13 Dec. 2013]

[Hav12b] Marijn Haverbeke. 2012. *Faking an editable control in browser JavaScript*. URL=http://marijnhaverbeke.nl/blog/browser-input-reading.html [Date accessed: 13 Dec. 2013]

[Lain12] Markku Laine, Denis Shestakov, Petri Vuorimaa. 2012. *XFormsDB: an extensible web application framework built upon declarative W3C standards*. In: SIGAPP Appl. Comput. Rev. 12, 3 (September 2012), pp. 37-50. DOI=10.1145/2387358.2387361

[Nem12] Cristina Nemeş, Marius Podean, Lucia Rusu. 2012. *XRX: The Implementation Process under XRX Architecture*. In: WEBIST, SciTePress (2012), pp. 103-109

[Sten12] Pontus Stenetorp, Sampo Pyysalo, Goran Topic, Tomoko Ohta, Sophia Ananiadou, Jun'ichi Tsujii. 2012. *BRAT: A Web-based Tool for NLP-Assisted Text Annotation*. In: Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics (April 2012)

[Col11] Timothy W. Cole, Myung-Ja Han. 2011. *The Open Annotation Collaboration Phase I: Towards a Shared, Interoperable Data Model for Scholarly Annotation*. In: Journal of the Chicago Colloquium on Digital Humanities and Computer Science, vol. 1, no. 3 (2011)

[Cout11] Alain Couthures. 2011. *JSON for XForms*. In: XML Prague 2011 – Conference Proceedings, Univerzity Karlovy v Praze (2011), pp. 13-24

[Dipp11] Stefanie Dipper, Martin Schnurrenberger. 2011. *OTTO: A Tool for Diplomatic Transcription of Historical Texts*. In: Human Language Technology. Challenges for Computer Science and Linguistics, Lecture Notes in Computer Science vol. 6562 (2011), pp. 456-467

[Ett11] Thomas Etter, Peter M. Fischer, Dana Florescu, Ghislain Fourny, Donald Kossmann. 2011. *XQuery in the Browser reloaded*. In: XML Prague 2011 – Conference Proceedings, Univerzity Karlovy v Praze (2011), pp. 191-210

[Gor11] Aleksejs Goremikins, Henry S. Thompson. 2011. *Client-side XML Schema validation*. In: XML Prague 2011 – Conference Proceedings, Univerzity Karlovy v Praze (2011), pp. 1-12

[Kay11] Michael Kay. 2011. *XSLT in the Browser*. In: XML Prague 2011 – Conference Proceedings, Univerzity Karlovy v Praze (2011), pp. 125-134

[Kilp11] Pekka Kilpeläinen. 2011. *Checking determinism of XML Schema content models in optimal time*. In: Information Systems, vol. 36, Issue 3 (May 2011), pp. 596–617. DOI=10.1016/j.is.2010.10.001

[Lain11] Markku Laine, Denis Shestakov, Evgenia Litvinova, Petri Vuorimaa. 2011. *Toward Unified Web Application Development*. In: IT Professional vol. 13, no. 5 (September 2011), pp. 30-36. DOI=10.1109/MITP.2011.55

[Mil11] R. Alexander Milowski. 2011. *Efficient XML Processing in Browsers*. In: XML Prague 2011 – Conference Proceedings, Univerzity Karlovy v Praze (2011), pp. 135-148

[Toff11] Giovanni Toffetti, Sara Comai, Juan Carlos Preciado, Marino Linaje. 2011. *State-of-the Art and trends in the Systematic Development of Rich Internet Applications*. In: J. Web Eng. vol. 10, no. 1 (March 2011), pp. 70-86

[Zhuan11] Canwei Zhuang, Ziyu Lin, Shaorong Feng. 2011. *Insert-friendly XML containment labeling scheme*. In: Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM '11). DOI=10.1145/2063576.2063989

[Bol10] Michael Bolin. 2010. *Closure: The Definitive Guide*. O'Reilly, Sebastopol (September 2010)

[Gruen10] Christian Grün. 2010. *Storing and Querying Large XML Instances*. URL=http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-127142 [date accessed: 13 Dec. 2013]

[Hees10] Ralf Heese, Markus Luczak-Rosch, Radoslaw Oldakowski, Olga Streibel, Adrian Paschke. 2010. *One Click Annotation*. URL=http://ceur-ws.org/Vol-699/Paper4.pdf [date accessed: 13 Dec. 2013]

[Kay10] Michael Kay. 2010. *A Streaming XSLT Processor*. In: Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, vol. 5 (2010). DOI=10.4242/BalisageVol5.Kay01

[OCon10] Martin F. O'Connor, Mark Roantree. 2010. *Desirable properties for XML update mechanisms*. In: Proceedings of the 2010 EDBT/ICDT Workshops (EDBT '10). DOI=10.1145/1754239.1754265

[Pham10] Thuan Pham, Mark Kochanski. 2010. *Model-View-Controller Design*. URL=https://faculty.washington.edu/markk/sites/default/files/CSS555_Model-View-Controller_Design.pdf [date accessed: 13 Dec. 2013]

[Pohj10] Mikko Pohja. 2010. *Comparison of common XML-based web user interface languages*. J. Web Eng., vol. 9, no. 2 (June 2010), pp. 95-115

[Cayl09] Hugh A Cayless. 2009. *Linking Text and Image with SVG*. In: Kodikologie und Paläographie im digitalen Zeitalter - Codicology and Palaeography in the Digital Age. Schriften des Instituts für Dokumentologie und Editorik, 2. BoD, Norderstedt, pp. 145-158

[Kuus09] Janne Kuuskeri, Tommi Mikkonen. 2009. *Partitioning web applications between the server and the client*. In: Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09). DOI=10.1145/1529282.1529416

[Hev09] Miško Hevery, Adam Abrons. 2009. *Declarative web-applications without server: demonstration of how a fully functional web-application can be built in an hour with only HTML, CSS & Javascript Library*. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09). DOI=10.1145/1639950.1640022

[Lam08] Tak Cheung Lam, Jianxun Jason Ding, Jyh-Charn Liu. 2008. *XML Document Parsing: Operational and Performance Characteristics*. In: Computer, vol. 41, no. 9 (September 2008), pp. 30-37. DOI=10.1109/MC.2008.403

[Li08] Changqing Li, Tok Wang Ling, Min Hu. 2008. *Efficient updates in dynamic XML data: from binary string to quaternary string*. In: The VLDB Journal, vol. 17, Issue 3 (May 2008), pp. 573-601

[McCr08] Dan McCreary. 2008. *XRX: Simple, Elegant, Disruptive*. URL=http://www.oreillynet.com/xml/blog/2008/05/xrx_a_simple_elegant_disruptiv_1.html [date accessed: 13 Dec. 2013]

[Fan07] Wenfei Fan, Gao Cong, Philip Bohannon. 2007. *Querying xml with update syntax*. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07). DOI=10.1145/1247480.1247515

[Fons07] José M. C. Fonseca, Ignacio M. Prendes, Javier Soriano, Juan J. Hierro. 2007. *Declarative Models for Ubiquitous Web Applications*. URL=http://www.w3.org/2007/02/dmdwa-ws/Papers/jose-m-c-fonseca.html [date accessed: 13 Dec. 2013]

[Gou07] Gang Gou and Rada Chirkova. 2007. *Efficient algorithms for evaluating xpath over streams*. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07). DOI=10.1145/1247480.1247512

[Hav07] Marijn Haverbeke. 2007. *Implementing a Syntax-Highlighting JavaScript Editor–in JavaScript*. URL=http://codemirror.net/1/story.html [date accessed: 13 Dec. 2013]

[Honk07] Mikko Honkala. 2007. *Web User Interaction–a Declarative Approach based on XForms*. In: Publications in telecommunications software and multimedia (2007)

[Mikk07] Tommi Mikkonen, Antero Taivalsaari. 2007. *Web Applications–Spaghetti Code for the 21th Century*. URL=http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8586 [date accessed: 13 Dec. 2013]

[Fowl06] Martin Fowler. 2006. *GUI Architectures*. URL=http://martinfowler.com/eaaDev/uiArchs.html [date accessed: 13 Dec. 2013]

[Khai06] Aye Khaing, Ni Lar Thein. 2006. *A Persistent Labeling Scheme for Dynamic Ordered XML Trees*. In: International Conference on Web Intelligence (December 2006). DOI=10.1109/WI.2006.23

[Pemb06] Steven Pemberton. 2006. *XForms Quick Reference*. URL=http://www.w3.org/MarkUp/Forms/2006/xforms-qr.html [date accessed: 13 Dec. 2013]

[Zhan06a] Jimmy Zhang. 2006. *Simplify XML processing with VTD-XML*. URL=http://www.javaworld.com/javaworld/jw-03-2006/jw-0327-simplify.html [date accessed: 13 Dec. 2013]

[Zhan06b] Jimmy Zhang. 2006. *Cut, paste, split, and assemble XML documents with VTD-XML*. URL=http://www.javaworld.com/javaworld/jw-07-2006/jw-0724-vtdxml.html [date accessed: 13 Dec. 2013]

[Zhao06] Li Zhao, Laxmi Bhuyan. 2006. *Performance Evaluation and Acceleration for XML Data Parsing*. URL=http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.5330 [date accessed: 13 Dec. 2013]

[Mur05] Makoto Murata, Dongwon Lee, Murali Mani, Kohsuke Kawaguchi. 2005. *Taxonomy of XML schema languages using formal language theory*. In: ACM Trans. Internet Technol. vol. 5, no. 4 (November 2005), pp. 660-704. DOI=10.1145/1111627.1111631

[Li05] Changqing Li, Tok Wang Ling. 2005. *QED: a novel quaternary encoding to completely avoid re-labeling in XML updates*. In: Proceedings of the 14th ACM international conference on Information and knowledge management (CIKM '05). DOI=10.1145/1099554.1099692

[Silb05] Adam Silberstein, Hao He, Ke Yi, Jun Yang. 2005. *BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data*. In: 21st International Conference on Data Engineering, 2005. Proceedings (April 2005), pp. 285-296. DOI=10.1109/ICDE.2005.29

[Lind04] Tancred Lindholm. 2004. *A three-way merge for XML documents*. In: Proceedings of the 2004 ACM symposium on Document engineering (DocEng '04). DOI=10.1145/1030397.1030399

[Steel04] Oliver Steele. 2004. *Web MVC*. URL=http://osteele.com/posts/2004/08/web-mvc [date accessed: 13 Dec. 2013]

[Bart03] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari. 2003. *Streaming XPath Processing with Forward and Backward Axes*. In: 19th International Conference on Data Engineering, 2003. Proceedings (March 2003), pp. 455-466, DOI=10.1109/ICDE.2003.1260813

[Dub03] Micah Dubinco. 2003. *XForms Essentials*. URL=http://xformsinstitute.com/essentials/browse/ [date accessed: 13 Dec. 2013]

[Nic03] Matthias Nicola and Jasmi John. 2003. *XML parsing: a threat to database performance*. In: Proceedings of the twelfth international conference on Information and knowledge management (CIKM '03). DOI=10.1145/956863.956898

[Trew02] Shari Trewin, Gottfried Zimmermann, Gregg Vanderheiden. 2002. *Abstract user interface representations: how well do they support universal access?*. SIGCAPH Comput. Phys. Handicap. 73-74 (June 2002), pp. 77-84. DOI=10.1145/960201.957219

[Tsai98] Michael Tsai. 1998. *WYSIWYG: Is it What You Want?*. URL=http://www.atpm.com/4.12/page7.shtml [date accessed: 13 Dec. 2013]

[Burb87] Steve Burbeck. 1987. *Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)*. URL= http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html [date accessed: 13 Dec. 2013]