Faculty of Mathematics, Physics and Informatics, Comenius University
Department of Computer Science

# WYSIWYG XML modification driven by schema

Diploma Thesis
April 2004

**Author:** Martin Vyšný
**Thesis advisor:** RNDr. Vanda Hambálková

# Acknowledgement

I would like to thank my parents for their support during the preparation of the thesis and my studies altogether. Concerning the thesis I would like to thank Tibor Vyletel for his ideas and suggestions which helped me to form the overall design, Prof. RNDr. Branislav Rovan PhD., RNDr. Vanda Hambálková and Mgr. Marek Nagy for many helpful hints and feedback.

# Contents

# Appendices

# Introduction

# 1

In today's heterogenous data processing environment, it is important to achieve simple and fast data processing and retrieval. To achieve this requirement, data must be stored and transported in well-defined, unified way. Several data storage formats were developed, however, they were not universal, mutually compatible and they mostly lack simple and powerful ways to retrieve data. Finally, in 1998 a new format was introduced, named XML, based on SGML technology. This thesis proposes an approach to storing and working with documents stored in XML.

The structure of this thesis is as follows. This chapter describes the advantages of a new approach to document modification and storage, and several free open-source tools dealing with the required functionality. The basic concept of WYSIWYG XML editor is laid in chapter 2: WYSIWYG XML Editor Concept, where various technologies are compared and described. In order to be able to modify the document in a WYSIWYG style we formed the basic definitions and terms, described in chapter 3: Document Modification Concept. Finally, the developed algorithms allowing us to do the WYSIWYG modification are introduced in chapter 4: Schema.

## 1.1. Motivation

The idea behind this thesis is to prove, that it is possible to build WYSIWYG editor*, capable of editing an XML. No similar works have been published yet, and such editors have been released only recently. However, they are mostly not pure WYSIWYG: we can see the result, but we can't edit it in a WYSIWYG fashion - we modify the XML source code instead. For such a universal format, a universal editor is missing. Our task is therefore to identify and solve such primary issues, that would allow us to build such editor.

### 1.1.1. Document semantics

Most of document formats were developed with presentation of document in mind. Presentation says clearly and unambiguously, how the document must be rendered on an output device, such as monitor or printer. However, for indexing and search purposes, this storage method is insufficient: we can't for example research all documents written by a particular author without knowing which string in the document forms the author's name. This is the major drawback of all presentation-oriented document formats, for instance the HTML document format. Lately, HTML have been extended to hold some semantic values, such as keywords for example, however, we

* What You See Is What You Get, this term originally meant that we are capable of getting the document, we are currently editing on-screen, to a printer. Its meaning had been slowly widened to following: we are capable of editing the graphical representation of document, in an user-friendly environment, regardless of their internal structure and representation

can't describe virtually unlimited set of semantics with just a limited set of elements, and keep this description simple. Thus, a new document representation, named XML (extended markup language) was born. It has similar syntax as HTML with some features:

- Elements do not have predefined presentation
- It may hold elements with virtually any name

Hence we are free to say which elements should be contained by the XML and how should the XML structure look like. Because we aren't limited by the presentation definitions, we can start to develop the semantic-layered documents.

### 1.1.2. Advantages

The approach of storing semantic values in a document has following advantages: if we define presentation rules for each element we can render this document in various ways: on a graphical device, such as monitor or printer, we can send it to an electronical human-voice synthesizer and let it read only important parts with various pronounciation. Because we aren't limited by finite set of elements, we may develop our own XML that exactly meets our needs. There are already well-defined XML structures, registered as standards for storing mathematical formulae, 3D scenes, chemical formulae or even musical notations. We can see that XML is truly a variable format capable of holding virtually any data.

### 1.1.3. Document processing tools

A definition of document storage itself is not sufficient. We require that:

1. A document can be easily read (deserialized) and prepared for automatized processing,
2. It can be easily transformed to a document with different structure that may meet our needs better,
3. It can be rendered to a widely-used output device or format,
4. Document's valid structure should be easily describable.

XML meets all these requirements by introducing a standardized object model holding the XML in a computer memory and featuring open-source, free readers in virtually all programming languages, capable of deserializing XML into this object model. It also introduces standardized language for XML transformation, several languages for document presentation, several languages for defining XML valid structure and free tools to use these languages in various programming languages.

Therefore we can say that XML is a full-valued tool for document processing and storing. Thus this diploma thesis is based on this standard.

## 1.2. Thesis goals

We tried to lay basis of a universal extendable document editation framework using new open-source technologies verified in practice. The main aim of this thesis is to design and

implement the following:

- To allow the document to be edited following the document validity descriptor language (from now on referred to as a schema),
- To design and implement main transformation chain,
- To specify overall design of the framework.

# WYSIWYG XML Editor Concept 2

The editor must be able to render the document on the page due to presentation specification, and edit this graphical representation, instead of XML source code. We can split the functionality into following modules:

- Transformer - transforms a semantic document to a presentation document. It should be responsible for validity of the result document, however this rule is hard to fulfil: checking the XSLT script, if it produces valid result is beyond scope of this thesis.
- Renderer - renders the presentation document onto given graphical canvas. It should try to render even if the document isn't valid, or it may safely throw an error.
- Editor - tries to achieve WYSIWYG editation of a semantic document.
- Schema - manages allowed semantic document structure. The document must be valid in order to get transformed, otherwise the transformation result may not be valid.

## 2.1. XML introduction

This chapter describes basics of the XML file. For exact definition please look at [XML98].

As we said before, XML is a universal data format capable of holding description of virtually any data. It it a tree structure having several types of nodes (following names are used in DOM terminology):

- `Document` - always a root node, in the context of the XML tree. It may hold at most one child element which is called a root element - although it is not a root node, it is a topmost element in the document.
- `Element` - builds a document structure. Only this type of node can contain attributes.
- `Attr` - an attribute, defines additional properties of element. Although a node, it is not a part of document tree. Thus it is not amongst children of element - they form a separate list of nodes in an element. Every attribute has a text value.
- `Text` - holder of textual value, also defines textual value of an element.
- `CDATASection` - shorthand for character data, it can also hold textual value as a text node, however it can hold virtually any string of characters, even a binary data.
- `Comment` - holds comments present in the XML file.
- `ProcessingInstruction` - shorthand for processing instruction. It is intended to hold hints (or even executable code, for example javascript) useful when processing the XML.
- `Entity` - children of the entity define its body. It is contained in special `DocumentType` node (if the document has one).
- `EntityReference` - this node can be anywhere in document, but it must be descendant of root

element. It defines the reference to entity, and the subtree rooted in this node is equal to the subtree rooted in adequate `Entity` node.

- `DocumentType` - defines document type, it can be a child of `Document` only.

Each element and attribute has its own name, the qualified name or qname. Qualified name consists of two strings: a namespace string, and a local name string. Namespace string identifies relation of element or attribute to some XML language standard. For example, all elements, defining mathematical formula in MathML standard language have a namespace string equal to the "http://www.w3.org/1998/Math/MathML" string. Local name defines the name of the element in context of its namespace. We shall write prefixed qname as foo:bar and namespaced qname as {http://foo.sk/foo}bar. The "{}bar" notation specifies a `null` namespace.

If an element has empty (`null`) namespace, then it is said that the element does not have any namespace. This is a difference between elements and attributes: if an attribute has empty namespace, it inherits the namespace from its element. This raises some ambiguity: an attribute can belong to one context with two different namespaces: empty namespace and owner element's namespace. Therefore, in our document model, we forbid the attribute to have same namespace as its owner element.

Proper using of namespaces allows us to mix various XML language standards in one document. For example we may have some XML language describing pure text documents, such as DocBook, containing some mathematical formulae described in pure mathematic language such as MathML. This mixing greatly improves the power of XML language because each language can concentrate on describing its scope of interest perfectly and can leave things outside its scope to other languages. However, it also brings some complication, as we shall see.

Let's define terms used in this thesis. Nametree is a maximal coherent tree, where all elements have the same namespace (Note that it can contain attributes with different namespace). We can see that XML document is built from some disjunctive nametrees. Nameroot is a root element of some nametree. Root element of document is always a nameroot.

### 2.1.1. XML object representation

In order to work with XML document in Java environment, the XML document must have an object representation. We must be separated from serialized XML representation (for example, when stored on disk). A parser must be available to convert the document from this serialized state to object representation. In this thesis we shall use the Xerces parser.

There are generally two kinds of document representation: the DOM (Document Object Model) representation where the whole document is loaded into the memory and represented as a tree. The second kind of representation is the SAX event model, that walks over the XML document and fires events in an in-order ordering, such as:

- start of element: an element was found. Next its children will be examined.
- end of element: all descendants were enumerated.

SAX works as an in-order enumerator of all tree nodes. The main difference between DOM

and SAX is that when using SAX we don't have in-memory representation of whole document, unless we store it somewhere.

## 2.2. Technology decision

There are basic differences between technologies we are going to use - they may have different rendering power and complexity, speed, and overall philosophy. Therefore, we must decide in this initial stage which technology will be used and which technology will be build upon.

### 2.2.1. Programming environment

We assume that all programming languages have all necessary technologies already implemented, so this would not be the main issue in picking the programming environment and language.

#### 2.2.1.1. C / C++

The only feature of this solution is its speed and low memory consumption. However, these advantages will eventually fade away as the computational speed and amount of memory will graduate. It is not natively multiplatform (although the code may be made compilable on different platforms). It has no visual components framework, nor a garbage collector.

#### 2.2.1.2. Mozilla and JavaScript

This solution offers great pluginability - the plugins can be coded in both javascript and C++. It even contains built-in WYSIWYG editor, capable of editing a XML rendered by CSS3 technology. It has already implemented ability to print documents and export them to PDF file. However, CSS3 language without prior transformation isn't sufficient - it is not capable of generating a table of contents for example. Also, the editor isn't fully WYSIWYG - it prints the document to a page medium, but the editation displays the document onto one page, with infinite height. This could move footnotes to the end of the document, for example.

#### 2.2.1.3. Eclipse and Java

The Eclipse Project is an open source software development project dedicated to provide a robust, full-featured platform for development of highly integrated tools. It offers great GUI capabilities through its GEF framework, however, it doesn't offer so much as Mozilla because it is far more universal. It accepts plugins programmed in Java which is a robust multiplatform language with many open-source tools.

### 2.2.2. Presentation technology

As we already said, CSS3 without prior transformation isn't strong enough to generate table

of contents. Even with transformation support, this technology has no page layout capabilities, as footmarks, page headers, etc. We need more professional typesetting tool, like [XSLFO01] (XSL-FO) processor. This processor can be used as our rendering plugin, therefore we must use the transformer + renderer model, as described later.

### 2.2.3. Conclusion

The requirement is to use the model of transformer and renderer, thus we cannot use Mozilla: its WYSIWYG editor is capable of working only with CSS3, with no transformations. Therefore, to implement this project the Eclipse and Java combination was chosen. For transforming purposes we chose the XSLT language because it is intended only for transforming and it is powerful enough to create virtually any output.

## 2.3. Rendering

There is no universal language that describes how to display an XML, thus there is no universal renderer either. We could have a renderer capable of rendering some XML language or a set of languages. However, we can create our own XML language and it would require to create new renderer for this language. A better idea may be to create one renderer capable of rendering the HTML for example, and an independent transformer transforming our new language into the HTML. This approach will be simplier only if creation of a new transformer is much easier than creation of a new renderer. XML complies this requirement - there is quite simple, yet powerful transformation language, called XSLT.

We can describe the rendering chain in following steps (layers):

1. Semantic layer - we have a source XML document. This document can have mixed semantic and presentation contents or no semantic contents at all, however, it should have a pure semantic structure to fully divide the semantic and presentation aspect.
2. Transformation layer - source document gets transformed to a document that the renderer is capable to display.
3. Presentation layer - we have a document directly renderable by some renderer. Again, this document may also contain non-presentational elements but it must be renderable without any modifications.
4. Render layer - the renderer takes a document from presentation layer and prepares it for displaying in a graphical device. It involves computation of absolute coordinates for example.
5. Display layer - takes the result of render layer and displays it on a graphical device.

### 2.3.1. Transformation layer

We will use the XSLT (XML Stylesheet Language*) for transformation, and a Java library

* the name 'stylesheet' is derived from its primary function to transform semantic document into presentation document, thus giving it a presentation style

called Xalan ([XAL260]) to load these scripts and to do the transformation. This library is a black box - we can use it through the TRaX ([TRaX]) interface for transformation but we do not see inside the library and we have no control over the transformation process.

Each transformation script should work for exactly one XML language - for exactly one namespace. However, the semantic document may contain multiple namespaces, thus multiple scripts may be used to process the document. There is no simple way to create one transformation script from multiple scripts. All XPath expressions would have to be modified to work only with one nametree for example. It is simplier to break the document to disjunctive nametrees and let each nametree be transformed by appropriate transformer. To build transformed document from all transformation results we must link foreign nameroot placement in source nameroots and retain these links in a result tree. This is done by introducing a special mark element that marks the original position of the foreign nameroot in every nametree. XSLT script must be modified to copy this element everywhere where the original foreign nameroot would be copied. Note that transformation can produce virtually any output, even with multiple namespaces, therefore a nametree does not neccesarily transform into one nametree.

We do not know without further XSLT analysis what is changed in the presentation document as a response to a semantic document modification. Thus every atomic document operation must be followed by a transformation in order for the operation result to be displayed. Transformation is a time-critical operation and it must be executed as fast as possible. Every speedup optimalisation is essential even at the cost of higher memory usage.

Breaking the semantic document to nametrees everytime before a transformation is not optimal. Even a slight change in the semantic document may require a transformation of the document, requiring the document to be broken into a nametrees after each modification. Hence we should hold these nametrees always in memory and all the changes in the semantic document must be reflected to the nametrees. To speed up the transformation even more we shall remember transformation results for each source nametree, so we shall have to transform only the changed nametrees. This solution triples the memory used to store a document.

## 2.4. Backwards identification system

To allow WYSIWYG editing of document graphical presentation directly in the canvas we must first identify the original data placement in the semantic document. We could use the SAX to feed the transformer with elements, watch the transformed result being produced and trace links between source and result elements (what the transformer have produced for each consumed element). However, this approach is neither effective nor correct. Let's suppose we select all elements in document with an XPath expression "`//*`". The transformer must read the whole source to compute this selection correctly and build its internal DOM representation, all this without any output. The internal document is not necessary when the source is given in a form of a DOM model. Moreover, we do not have a simple element-subtree link therefore we can't trace the links properly.

Let's give each node an ID. We say that the node is denoted by its ID. This ID value must

be part of the XML document model in order to be visible to the stylesheet. In other words it must be a node. This ID node should be contained directly in the node that it denotes to simplify the ID processing in a stylesheet. An ideal node type is an attribute - it does not belong to a node's child nodes, therefore it doesn't change their position number. The stylesheet itself must be modified to transport these IDs correctly.

However, only an element can contain an attribute, and we must be able to identify all types of nodes. We could encapsulate all non-element nodes in a special element but that would greatly complicate all XPath expressions: for example each "`*`" expression would have to be replaced by "`*[not(self::em2p:idref)]`". Instead, we let the stylesheet compute ID of these nodes:

- If the node is an element, we simply take the value of its ID attribute. Let this ID attribute value always be an integer number.
- If the node is not an element nor an attribute then it is a child of some element and it has an order number. Hence, we may compute its ID from the ID of its parent and its zero-based position. Separator will be a semicolon (`;`).
- Attributes are not ordered so we must use their qname instead of their position and separate it by a `@` character (this character is used in XPath to mark attributes).

The computed ID can be for example given as an attribute to the element created from the source node denoted by this ID.

The order number of a node must be computed by an XPath expression "`count(preceding-sibling::node())`" rather than computing a zero-based position (`position() - 1`). The main difference is that the `position()` function counts position of the node relative to the context nodeset, but the `preceding-sibling` axis selects all preceding nodes in original document, regardless of the context nodeset. For some examples on how to modify the transformation, please look at appendix A: XSLT Modification Hints.

## 2.5. Editor namespace

To make our reserved element and attribute placeable into the XML content, we must define their namespace. We reserve a special namespace "http://www.uniba.sk/euromath2/reserved", and its reserved prefix "em2p:". Nor this namespace, nor the prefix can occur in original XML document. We can now define names for our new elements and attributes. The identification attribute qname shall be em2p:id and mark element qname shall be em2p:mark, both having namespace of EuroMath2*. The em2p:mark element has the following content:

- The em2p:id attribute. Its value is equal to ID of nametree's nameroot, that is represented by this mark element.
- Its string value is equal to the namespace of the nameroot that the em2p:id attribute denotes.

---

* This diploma thesis aims at developing the editor named EuroMath2, hence from now on we shall refer to this editor as EuroMath2.

# Document Modification Concept 3

## 3.1. Entities

Entities are constants in XML. To process the XML correctly, entity references must be replaced by the actual entity value. Each entity is denoted by a unique name. If more than one entity with the same name is defined then the last entity definition must be used. However, this behaviour is implementation-specific: the XML definition ([XML98]) allows parser to throw an error and deny the further parsing of document. EuroMath2 uses this 'last entity definition' strategy. There is one exception: XML standard defines five entity names that cannot be redefined: `amp` denoting an ampersand character, `lt` denoting less-than, `gt` as greater-than, `quot` as quotes and `apos` as an apostrophe. An attempt to redefine these five entities will result in error as well as usage of undeclared entities.

There are two types of entities: parsed and unparsed. Parsed entities allow us to define parts of DTD and reuse them, but only in context of DTD. XML can refer only to unparsed entities. There are two types of unparsed entities: internal and external. For exact definition please look at [XML98]. In short, external entities contain a URL address pointing to another XML file. Therefore, entities can contain a subtree.

The content of entities in current DOM model level is generally unmodifiable - it is defined to be readonly in both Entity and EntityReference nodes. The current Xerces implementation allows us to do some hacking: we can clear the read-only flag and modify the entity contents. However, when the document is serialized, these changes are not written even if modified entities are defined directly in the document. Therefore, entity definitions are unmodifiable in the current EuroMath2 implementation.

To determine whether an entity is modifiable, we must know the URL of file where the entity is located. All non-locally stored entities (having a URL that uses HTTP or FTP transfer protocol), and entities stored locally in EuroMath2 directory must be read-only. Hence, the only entities defined in the document file (except the five reserved XML entities) and entities with local URLs (for example URLs beginning with "file:/") except EuroMath2 locally cached entity files can be modified.

## 3.2. Document loading

In order to edit a document, it has to be loaded into a memory DOM representation. Following steps are to be taken (When an unrecoverable error occurs in any of these steps then the

document loading shall fail):

1. Load the document into a DOM representation. During this step, we must also load all entities known to XML and redirect URLs of entity files onto locally stored files when the redirection is defined (for example in the configuration file). Failure to load and parse any of the required files results in an error.
2. Build list of all namespaces used in the document.
3. Initially process the document - add ID attributes to all elements, unify all namespace prefixes and split it into nametrees.
4. For each namespace the presentation layer (XSLT script) must be picked, either from the configuration file or manually by a user. If no script is found, a universal script can be selected - it displays a list of all nodes present in the document using XSL-FO language ([XSLFO01]).
5. For each namespace the schema must be picked. Schema must be present in order for the document modification to be schema-driven. Therefore, failure to load schema for some namespace must result in error.
6. Check the document for validity. Any validity mismatch results in error unless we can fix the problem by modifying the document. Complex document repair is beyond the scope of this thesis.
7. Document is transformed, then rendered onto the screen for the first time. Again, any transformation or rendering problems shall result in error.

When the algorithm is finished, the document is fully prepared for further modification.

## 3.3. Prefix unification

Prefixes are shorthands used to specify namespace in qname. They are defined via the `xmlns` attribute, in the form of `xmlns:prefix`. The local name 'prefix' denotes the name of prefix, and attribute value is appropriate namespace. This prefix is known to all descendants of the element (including this element) where it is defined (for details please see [XNS99]). Two qnames are equal if and only if their local names are equal and namespaces are equal (equal in this context means simple character by character string comparison). Hence two qnames may be equal even if they have different prefixes.

We see that prefixes do not define structure nor contents of XML, therefore they can be unified. When prefixes are unified the prefix mappings for all namespaces must be defined in the root element of an XML only, and for each namespace exactly one prefix must be defined. This solution has several advantages: 1) we can simply replace reserved `em2p` prefix*, 2) we can distribute all prefix definitions to root element of each nametree in the splitted representation to avoid problems with undeclared prefixes. The process of unification follows this algorithm:

1. Create a namespace-prefix mapping for each namespace in the document. If more than one

* We do not have to unify XML-reserved prefixes - they must be used solely for their defined purpose. When these prefixes are improperly used, an error is thrown at the time when the document is loaded into memory - the unification phase is not executed.

prefix is mapped to a namespace, then use any currently unused valid prefix (all prefixes beginning with `xml` and EuroMath2 prefix `em2p` are reserved and must not be used as common prefixes).
2. Remove all `xmlns` definitions from the document.
3. If no prefix was defined in original document for some namespace then use the prefix predefined in the configuration file, if it is unused.
4. If still no prefix is found then derive it from the namespace string (EuroMath2 does so by returning the last non-empty string not separated by characters illegal in the prefix name, such as slash or colon). Add some number if this prefix is still not unique.
5. For each mapping create prefix definition in root element. The `null` namespace shall be unprefixed thus it does not need to be declared.

We cannot unify prefixes in the body of external entities - they are read-only. They are allowed to inherit prefixes from the document, from which they are referenced. Thus, prefix unification may corrupt this inheritance. We may hack read-only behaviour and unify their prefixes, however this unification would render them unreferencable in other documents if we allowed them to be saved. This can be corrected by creating the prefix definitions in all elements that are children of `EntityReference` node.

## 3.4. Elementary and atomic operations

In order to implement complex document modification, we must identify and implement the minimal set of all required operations. We say that a set of operations is complete when every possible operation can be performed by applying a sequence of operations from this set. The complete set with minimal cardinality is said to contain elementary operations. The following list composes such set:

- Inserting a new node amongst children of some element,
- Deleting a node,
- Inserting a new attribute into an element (although also a node, it doesn't belong to children of the node),
- Deleting an attribute,
- Modifying value of a non-element node - this involves changing textual value of an attribute, text, cdata and comment node, and changing the name and/or textual value of a processing instruction node.

Textual value of an element is defined to be the concatenation of textual values of all descendant `Text` and `CDATASection` nodes, in the node order. Therefore, its value is modified by inserting, deleting and modifying of text and/or cdata node, thus it is a complex operation. Deleting a simple node causes delete of subtree rooted in this node thus delete of a node is both elementary and complex.

The requirement to edit an XML in WYSIWYG style adds complications to this simple model. Before the transformation and rendering we must assure that XML is valid against its

schema - XSLT is designed to compute valid results for valid XML files only. The result of transformation of invalid XML is undefined. Let atomic operation be any operation that receives a valid document and returns a valid document. Atomic operation may be one elementary operation or a sequence of elementary operations, depending on the schema. While the atomic operation is in progress, we cannot transform the document, thus we cannot show the operation progress in a WYSIWYG manner. Hence a good wizard is needed to drive the user through the execution of an atomic operation.

We shall implement the following atomic operations (for more detailed description please see chapter 4: Schema):

- Creating an element - this may require creating multiple elements, depending on the schema.
- Deleting an element - this may also require deleting multiple elements or even creating elements. For example we can have a choice between two elements and we want to delete one of them.
- Enclosing/declosing nodes - enclose some consecutive nodes in an element or replace an element with its contents.
- Creating an attribute - due to the requirements on the schema, this requires creating this attribute only.
- Deleting an attribute - it requires deleting this attribute only.
- Moving a group of consecutive nodes
- Moving an attribute - the presence of the attribute cannot be claused thus we can delete the attribute and create it somewhere else. Hence this operation can be replaced with two atomic operations.
- Creating, modifying and deleting comment and/or processing instruction nodes - schema does not contain rules for these types of nodes therefore they can occur anywhere in the document and they can be modified without need of any further modification or validation.
- Modification of textual value - the text of the document is less important than the document structure, and the text can be created after the structure creation. This is because the schema requires text presence only in a special case (when there is no sibling element). Hence, modification of text will not trigger modification of elements and/or attributes although its presence may prevent us from inserting some elements. Moreover, we cannot say from the datatype object what kind of text it accepts thus we cannot help a user with typing a proper text. Therefore, text modification is an elementary operation - if a new text isn't valid, then it is just reverted back to the original text.

## 3.5. Description of main modules

The Transformer module must contain transformers (stylesheets) for each namespace present in the semantic document. If there are more than one stylesheets defined for one namespace, then user must choose the presentation he wants. There is currently no support for multiple consecutive transformations. All known transformer URLs must be defined in the configuration file.

The Schema module must also contain schema for all namespaces, however there is no need to pick from multiple schema: schema describes the correctness of the document according to the definition of document content. If the schema language is not sufficiently strong then the Java code based validator should be used instead. The only requirement is that it must implement the `Schema` interface. Class location or the URL of schema language must be defined in the configuration file.

Renderer module provides renderers for transformer output. If the document contains unknown namespace then we cannot render the document. One renderer should be built-in: the [XSLFO01] renderer [FOP0205] capable of rendering the http://www.w3.org/1999/XSL/Format namespace. Using this renderer we are capable of rendering the output to PDF.

Editor module must provide WYSIWYG editor for each used namespace. However, this is not so simple. Let's have a SVG (Scalable Vector Graphics) document. When transformed with identity transformation, it should render as graphical image, hence, a WYSIWYG editor allowing us to draw a line interactively should be used. We cannot use this editor when SVG is transformed to other format (for example, using the `SELECT_ANY_XML` constant to transform any document to XSL-FO in a way similar to Mozilla or IE) because we actually do not see the image itself. We cannot use it even when MathML expression is transformed into SVG because the editor is designed to edit SVG document only. Hence the editor should be picked in the following order:

1. The editor specifies two namespaces. It is able to edit the document part only when the semantic nametree namespace is equal to the first namespace and the presentation namespace is equal to the second namespace. This editor is able to have best WYSIWYG capabilities hence it should have the highest priority.
2. The editor specifies only one namespace, the namespace of the semantic nametree. It is designed to edit the original document hence it has a higher priority than the next editor.
3. The editor specifies only one namespace, the presentation namespace. It is able to edit the presentation regardless of the semantic document.
4. The EuroMath2 default universal editor, capable of editing any combination of namespaces. The editation is driven by appropriate schema.

## 3.6. Using identifiers

We have already described the backwards identification system, now we shall try to use it. The system of transporting IDs may vary between plugins but in order to work with default editor, it must comply to some rules. Of course if custom editor plugin is used, then it may use its own rules. The default editor expects the following:

- If the ID is lost in the process of transformation and/or rendering then the node from the semantic document with this ID is not visually selectable (we cannot guess which part of screen we have to highlight when the node is selected) thus it is not selectable by a mouse. However it may be possible to select such node, for example by keyboard navigation or directly in the XML tree.

- The text is edited inplace only if the correct font properties are known.

# Schema

# 4

The well-formedness property of an XML document describes how the document must be represented in a serialized form (for example a file) in order to be able to be processed. However it says nothing about its valid content, allowed element names and placement etc. This is the role for the schema language. There are various schema languages, perhaps the most known one is the DTD (Document Type Definition) language. DTD has a special position amongst the schema languages:

- It is the only language that allows us to define entities (this feature may be used in combination with another languages, for example the DTD may define entities and RELAX NG rules)
- It is unaware of XML namespaces: it may accept element mml:root but not math:root, even when both prefixes are bound to same namespace, and it accepts mml:root regardless of the namespace bound to the mml prefix.
- It does not have a support for datatypes
- It does not have the ALL rule (this rule allows us to specify for example any element ordering).
- It is not written in an XML format.

XML Schema language ([XSD01]) and TREX ([TREX]) languages was created to overcome these disadvantages. Because XML Schema is quite complicated language, RELAX language was introduced, followed by its successor RELAX NG ([RNG01]), combining advantages of both RELAX and TREX and inheriting simplicity of RELAX. To fully understand following text, we recommend to know at least basics of RELAX NG, because the terminology is used also in MSV Grammar.

## 4.1. MSV grammar model

Sun Microsystems built an universal validator capable of validating XML with all schema languages. This validator is named MSV (Multiple Schema Validator). Because schema languages are rather different and programming a validator for each language is not efficient, new language was introduced, mainly based on the RELAX NG language. We shall call this language the MSV grammar. MSV grammar is capable of represent all known schema languages. The process of validation is as follows:

1. MSV transforms the schema instance to a MSV grammar,
2. Validator validates the XML document using this MSV grammar schema representation.

MSV grammar is a grammar generating an XML tree. It may be represented as an oriented graph of rules, edge is oriented from a rule to its children. It is not necessary a tree and it may contain cycles, however each cycle must contain an element rule. It has exactly one root rule (it is connected only to outgoing edges). The grammar may contain following rules:

| Rule name | Marking | Child count | Description |
|---|---|---|---|
| Element | `<E>` | 1 | Generates an element, with possible element qnames from the `E.nc` set. Its child is a rule, which generates a tree rooted in this element. |
| Attribute | `@A` | 1 | Generates an attribute, with possible attribute qnames from the `A.nc` set. Its child is a rule, which defines the allowed textual content of the attribute. |
| Choice | `A \| B` | 2 | Content is generated by rule `A` or rule `B` (but not by both of them). |
| Interleave | `A ALL B` | 2 | The rule is similar to the Shuffle operation: sequence of roots of trees generated by rule A is mixed with sequence generated by rule B (ordering must remain unchanged). |
| Sequence | `A.B` | 2 | Sequence of roots of trees generated by `A` is followed by sequence generated by `B`. |
| Mixed | `MIX(A)` | 1 | Sequence of roots of trees generated by `A` is mixed (interleaved) by any sequences of characters. It is equivalent to `A ALL AnyString`. |
| OneOrMore | `A+` | 1 | The content is generated by applying the rule `A` once or more times. |
| Data | `data` | 0 | Generates textual string, belonging to the `data.type` set. This is generally equivalent to `DataOrValueExp` interface in MSV grammar object representation. |
| Epsilon | ε | 0 | Generates no contents. |
| AnyString | `data.*` | 0 | Generates any textual string. |

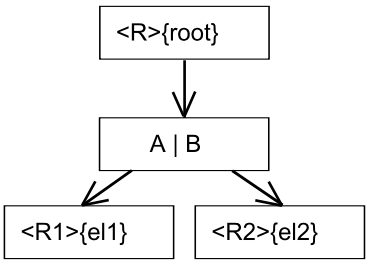


*Figure 4.1.1: This MSV grammar generates two possible XML documents: root element root containing one el1 or one el2.*

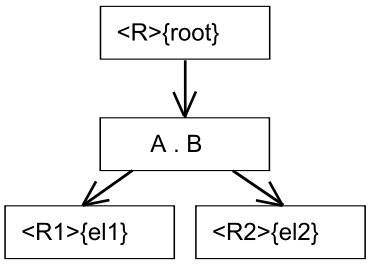


*Figure 4.1.2: This MSV grammar generates only one XML: root element root containing two children, the first one is el1 and the second one is el2.*

### 4.1.1. Nameclass

Each element and attribute rule has the nc set. The element rule is capable of generating an element whose qname belongs to this set, the same is valid for all attribute rules. For example one attribute rule can generate one attribute but with multiple qnames. The nc set is defined as a tree of objects; if the nameclass object accepts given qname then this qname belongs to the nc set and vice versa. The nameclass rule is built from the following rules:

| **Rule name** | **Marking** | **Child count** | **Description** |
|---|---|---|---|
| AnyNC | {*}* | 0 | Accepts anything. |
| LocalNC | {*}name | 0 | Accepts all qnames that have local name equal to 'name'. |
| NamespaceNC | {ns}* | 0 | Accepts all qnames that are from the specified nameclass. |
| SimpleNC | {ns}name | 0 | Accepts one qname, the {ns}name qname. |
| ChoiceNC | A \| B | 2 | Accepts when at least one child accepts. |
| DifferenceNC | A - B | 2 | Accepts when A accepts and B does not accept. |
| NotNC | A' | 1 | Accepts when A does not accept. |

We shall use the following notation of rules: <R>{{foo}q} denotes rule R with set nc containing only one element qname: {foo}q. The notation @R{qn, ..} denotes an attribute rule R whose nc set is any set containing qname qn. Any rule which has the nc specified this way must be a local rule. The set expression R.nc denotes the nc set of the rule R.

Let's define a local and foreign qname. We shall define these terms for elements first:

- Local qname is a qname with namespace equal to the namespace of the schema.
- Foreign qname is a qname that is not a local qname.

The definition for an attribute is slightly different as the definition for an element*:

- Local qname is a qname with namespace equal to the `null` namespace.
- Foreign qname is a qname that is not a local qname nor equal to the namespace of the schema.

We shall need special rules that allow us to place elements and/or attributes from another namespaces into the document. These rules shall be named entry points and notated as EP and @EP respectively. They are defined as follows:

- Rule <E> is EP when nameclass difference E.nc - {schema_namespace}* is not an empty set.
- Rule @A is @EP when nameclass difference A.nc - {schema_namespace}* - {}* is not an empty set.

In other words, EP (@EP) contains some foreign qnames also.

Let's put a restriction on all nameclasses: nameclass cannot accept an infinite number of local qnames (this shall be explained later in chapter 4.4.1.2.1: The acceptor model). We can split

* We disallowed to specify a namespace, equal to a namespace of the owner element

all rules into two groups:

- The local type: the nc set contains only local qnames (thus it must be a finite set).
- The EP type: the nc set may contain local qnames as well as foreign qnames and it may be infinite. However, it must contain finite number of local qnames.

### 4.1.2. MSV grammar properties

We shall use the following terminology:

- Rule `B` is child of rule `A` if and only if there exists an edge, connecting `A` and `B`, oriented from `A` to `B`.
- Rule `B` is descendant of rule `A` if and only if `B` is child of `A` or there exists rule `R` different from `A` and `B` such that `R` is descendant of `A`, and `B` is child of `R`.
- Rule `B` is parent of rule `A` if and only if `A` is child of rule `B`.
- Rule `B` is ascendant of rule `A` if and only if `A` is descendant of rule `B`.
- Rule `B` is direct descendant of rule `A` if and only if `B` is child of `A` or there exists rule `R` different from `A` and `B` such that `R` is not an element rule, `R` is direct descendant of `A`, and `B` is child of `R`.

In order to use the MSV grammar we must formulate its properties first.

**Lemma 1:** `A|B = B|A`, i.e. the `|` rule is symmetric.

**Lemma 2:** `A|A = A`, i.e. the `|` rule is reflexive.

**Lemma 3:** `A ALL B = B ALL A`.

**Lemma 4:** When A and B are rules for elements or attributes then `A ALL B = (A.B)|(B.A)`. When one of them is not element nor attribute then this may not be valid. For example `(<`$R_A$`>{A}+) ALL <`$R_C$`>{C}` is able to generate the `A.A.C.A` element sequence.

**Lemma 5:** `A.`ε ` = `ε`.A = A`.

**Lemma 6:** ε`+ = `ε.

**Lemma 7:** ε ` ALL A = A`.

**Definition 8:** Rule is ε-reducible if it is able to generate no nodes.

**Definition 9:** Rule is children-ε-reducible if it is able to generate no elements and text, regardless of the number of attributes generated.

**Lemma 10:** Rule `R` is children-ε-reducible if rule `R'`, created from rule `R` by replacing all direct descendant Attribute rules with ε, is ε-reducible.
**Proof:** If we replace all Attribute rules with ε then it cannot generate any attributes, hence if it is children-ε-reducible then it is also ε-reducible and vice versa.

**Lemma 11:** Both ε and `AnyString` are ε-reducible and children-ε-reducible.

**Lemma 12:** When both rules `A` and `B` are ε-reducible then `A ALL B`, `A+`, `A.B` are ε-reducible. When rule `A` is ε-reducible then `MIX(A)`, `A|R` is ε-reducible for any rule `R`.

**Definition 13:** An element or an attribute is required when its presence is required somewhere in

the document.

**Lemma 14:** An element E or an attribute A is required when its rule <$R_E$>{E, ..} (@$R_A${A, ..}) is required. Rule R is required if it is not a descendant of an ε-reducible rule. When there exists rule $R_{asc}$, ascendant of R, such that $R_{asc}$ is ε-reducible then R is optional regarding to the rule $R_{asc}$. If rule R is optional regarding to $R_{asc}$ then it is optional regarding to all ascendant rules of $R_{asc}$.

**Lemma 15:** When two rules generate the same content (in case of element and/or attribute rules their nc sets must also be equal) then they are equal. MSV unifies such rules into one rule, except for element and attribute rules - when rules are equal then they are represented by one graph node.

**Lemma 16:** It is not possible to have two different rules describing the same content. The only rule that allows such behaviour is the Concur rule of the TREX schema - it represents the 'intersection' of content generated by two children rules. However, this rule is not in the RELAX NG schema hence we shall ignore such rule.

**Definition 17:** Element or attribute has a context-free rule R if R is able to generate this element or attribute, and there is no other rule <$R_2$> such that $R_2$.nc contains the qname of the element/attribute. In other words we are able to identify the element/attribute rule only from the element/attribute qname. Please note that one rule can be context free for one element and non-context free for another.

**Lemma 18:** MSV graph can contain cycles hence some rules may be ascendants and descendants of themselves.

**Lemma 19:** All MSV graph cycles must contain an element rule - no schema language allows us to define schema violating this requirement.

**Lemma 20:** Let C be a cycle in an MSV graph. Then at least one rule from this cycle must be ε-reducible otherwise we would be forced to generate an endless XML document branch (lemma 19), what is illegal.

**Lemma 21:** The rule of choice, as the only rule, allows us to make ε-reducible rule $R_1|R_2$ from non-ε-reducible rule $R_1$ and any ε-reducible rule $R_2$. We see that $R_1$ is optional regarding to the $R_1|R_2$ rule, hence we shall call such choice rule the rule of optionality.

**Lemma 22:** The MSV grammar model allows us to condition the existence of attribute by an existence of element, and vice versa. For example rule <$R_E$>{E} > (<$R_A$>{A}.@$R_B${B}) | ε says that element E can have no children and attributes OR it must have element A and attribute B. DTD, XML Schema and RELAX does not allow us to impose such requirements, however RELAX NG is able to do so. This ability would complicate our algorithms hence we shall not allow such rule. The following must be true:

- Existence of an element does not set conditions on existence of any attribute,
- Existence of an attribute does not set conditions on existence of any element,
- Existence of an attribute does not set conditions on existence of any other attribute

### 4.1.3. Defining text

The `data` rule allows us to specify the type of textual value. For example the rule `<R>{year} > data(int)` states that the year element must contain only the textual value (no child elements nor attributes) and it must be an integer. However this rule is too strong in conjunction with other rules. For example, having the rule `data(decimal).data(int)` generating 3.146543, we are unable to say where the first datatype ends and the second one starts. Therefore the following is true:

**Axiom 23:** There must be no Intersection, Sequence, OneOrMore nor Mixed rule, such that all child rules are capable of generate some text (child is either a `data` rules or a direct ancestor of some `data` rule).

**Axiom 24:** When the element contains some elements then we cannot define the datatype for the text. It means that when mixing text with elements then the mixed text must be the `AnyString` rule.

However we are able to specify rule such as `<R>{A} > data(int) | MIX(<R1>{B}+)` which says that `A` can contain an integer number or any number of `B` mixed with any text. We must be careful when specifying such rules, for example in the rule `<R>{A} > data(int) | MIX(<R1>{B}+|ε)` the `A` element can contain zero `B` elements mixed with text thus it accepts `AnyString` and the datatype loses its effect.

The model is not as simple as it looks because we may allow to insert any text in one place and forbid any text in the another (for example the rule `MIX(<R1>{E}+).(<R1>{E}+)`).

### 4.1.4. The rule identification

In order to get an information about possible element content, we must be able to identify the rule for this element. Let the document be valid. An element can be generated only by element rule that has the element qname in its nameclass set. If the element has a context-free rule then this rule is the correct rule (because there is no other rule that is able to generate the element, and the element is generated exactly by this rule because the document is valid). Please note that this set shall not be empty because the document is valid hence every element must be able to be generated by some rule.

If the set contains more than one rule then the level two test must be performed. The main idea is to compare the path to the root of the document with the path to a root rule in an MSV grammar graph. Again, only the rule that has at least one correct path may be the valid rule. This algorithm may be optimized by remembering possible parent element rules for each rule*, thus we don't have to walk the whole MSV grammar graph.

If the set still contains more than one element rule then we shall try to validate the element content with each element rule in the set (the level three test). This is the strongest test currently implemented hence the next requirement on the schema is: it must not have two different rules that

---

* The MSV grammar object model does not contain references to parent objects

are direct descendant of same element rule, the intersection of their nameclass set is not empty and they are capable of generating the same element (with equal content).

This requirement is not strong enough to deny the rule change. Let there be two rules for an element with qname E (let this element have empty content): $\langle R_{E1}\rangle\{E\} > \langle R_{EC}\rangle\{E_C\}$ and $\langle R_{E2}\rangle\{E\} > \varepsilon$. These two rules are distinguishable, however when we insert $E_C$ into E then the rule for E may change. This would complicate the following algorithms thus we must prohibit the presence of such rules. We require that the grammar must not contain two rules that:

- Their nameclasses have an non-empty intersection, and
- The tree generated by first rule is a subtree of some tree generated by the second rule.

## 4.2. Multiple namespaces

The schema is primarily intended to describe the document with one namespace (it may safely describe multiple namespaces, for example by using custom name classes in the RELAX NG language, however we shall consider the schema languages with one namespace primarily). XML may contain multiple namespaces hence we must specify how multiple schema shall cooperate. Basically we can split the document into nametrees and validate them using appropriate schema file.

The schema must specify the place where the foreign elements and/or attributes can be inserted - the schema must contain foreign rules. However some schema languages have poor or no support for defining foreign rules (for example a DTD), hence we must be able to modify the MSV grammar model to make the schema aware of foreign elements/attributes. This may be done by replacing the child rule $R_C$ in all element rules with the rule $R_C$ `ALL ((@EP+|ε) . (EP+|ε))`. This replacement ensures that any foreign element or attribute can be inserted everywhere in the document.

Current implementation of the schema model expects that:

- There is one schema for each namespace present in the document hence a schema is not required to define elements from another namespace.
- Each element is validated by a schema belonging to the element namespace, thus there is no need to define content of foreign rules - appropriate schema will be used instead and the content of the foreign rule will be ignored.
- Only a root elements can be inserted in place of the foreign element rule (if the rule nameclass accepts the qname of such root element).
- When we are validating foreign attributes, we have to pick the correct attribute rule only from the attribute qname. Thus only attributes that have a context-free rule can be inserted into a foreign element.

## 4.3. Nameclass set representation

The first idea behind the creation of different name class set representation was the

requirement to get one (any) qname that belongs to given nameclass set*. In 90% of cases the nameclass will be an instance of the SimpleNC nameclass and our task is simple: just return qname represented by this nameclass. However some nameclasses can be pretty complicated, for example {*}* - ({schema_ns}* | {sioux_ns}*) used to extend DTD with EPs.

To get the qname we can use the Formal Languages approach to solve this problem: SimpleNC, LocalNC, NamespaceNC and AnyNC can be easily represented with the regular automaton (RA). When we encounter NotNC we will just convert given RA to accept its complement; that is also an RA. ChoiceNC creates RA by union of two RAs. Finally, DifferenceNC creates an RA by intersection of first RA and complement of second RA. As we see, NameClass can be represented by RA and thus simply analyzed. However, a lot of work is required in order to implement this algorithm: RA representation, union, intersection and conversion to deterministic automaton. Hence we'll try to find a simplier algorithm.

The Nameclass power is at most equal to the regular automaton, but in fact it is much weaker because we can't concatenate in any way. In fact we have some qnames sets and, to make a result set, we are using some set operations. Therefore we shall find some form of set, that is capable of holding correct result of all nameclass operations and is simple enough to be analyzed for a result.

### 4.3.1. The BCA set

Let's have an object containing three sets, each set containing any number of SimpleNC, LocalNC, NamespaceNC and/or AnyNC (the basic nameclasses). First set is called Base because it forms base set content, second is the Corrector because it corrects base contents and third is the AddBase set – its qnames are present in our set even if they are accepted by Corrector. This object shall be called the BCA set. Qname is accepted by BCA if and only if:

1. it is accepted by at least one name class from Base and it is not accepted by all name classes from Corrector, or
2. it is accepted by at least one name class from AddBase.

Hence, BCA is formed by union of all Base nameclasses followed by subtraction of all nameclasses from Corrector followed by union of all AddBase nameclasses.

**Definition 25:** Two nameclasses are equal if and only if they accept the same set of qnames.

**Lemma 26:** Let qn be any qname from some NamespaceNC nameclass. Then all other qnames must have same namespace as qn.
**Proof:** Straighforward from the NamespaceNC definition.

**Lemma 27:** Let qn be any qname from some LocalNC nameclass. Then all other qnames must have same local name as qn.
**Proof:** Straighforward from the LocalNC definition.

**Lemma 28 (Equality of basic nameclasses):** The following is true:

---

* This shall be required when we will compute the system of representants

- {*}* is equal only to {*}*
- {*}name is equal only to {*}name (the LocalNC rule can be equal only to a LocalNC rule. Local names must be equal)
- {ns}* is equal only to {ns}* (namespaces must be equal)
- {ns}name is equal only to {ns}name

**Proof:** Qname sets of all basic nameclass types are subset of {*}* nameclass. Only qname set of basic nameclass type {*}* is superset to {*}*. Therefore the qname sets are equal and that implies that {*}* is equal to {*}*.

**Lemma 29 (Intersection of basic nameclasses):** A intersected by B equals to A if and only if B equals to A and both A and B are of same basic type.

**Proof:** For the {*}* nameclass, {*}* intersected with {*}* gives {*}* which is equal to {*}*. Intersection of two SimpleNC nameclasses is a SimpleNC nameclass if and only if they are equal. Let A and B be both NamespaceNC (the proof is similar for LocalNC). In order for the intersection of A and B be non-empty, some qname must belong to both A and B, therefore using lemma 26 (or lemma 27 when A and B are LocalNC) both A and B are NamespaceNCs with same namespace and therefore equal.

**Definition 30 (NBCA - Normalized BCA):** Let Base.nc be a set of all qnames that are accepted by at least one nameclass from Base set (the same definition is valid for the Corrector and AddBase sets). We say that the BCA set is normalized when following rules are true:

1. If the Base set contains {*}* then it must be the sole nameclass in the Base set.
2. A nameclass must not be both in Base and Corrector. If such nameclass exists then it must be deleted from both Base and Corrector sets.
3. A nameclass must not be both in Base and AddBase. If so then it must be deleted from the Base set.
4. A nameclass must not be both in Corrector and AddBase. If so then it must be deleted from the Corrector set.
5. No nameclass from Corrector can be subset of union of the rest of nameclasses in Corrector. If so then it must be deleted.
6. No nameclass from AddBase can be disjunctive with all nameclasses from Corrector. If so then this nameclass must be placed into the Base set instead.
7. No nameclass in Corrector can accept qname that is not accepted by some Base nameclass. If so then the nameclass must be replaced by appropriate nameclass or nameclasses (or removed, if it is disjunctive with all nameclasses from Base). Each NamespaceNC from Corrector must be replaced by adequate SimpleNC for each LocalNC in Base. Similarly each LocalNC must be replaced for each NamespaceNC.
8. Each nameclass in AddBase must not accept qname that is not accepted by some Corrector nameclass. If so then it must be copied to Base (and replaced if it is not disjunctive with all nameclasses from Corrector, otherwise deleted from AddBase). Each NamespaceNC from AddBase must be replaced by adequate SimpleNC for each LocalNC in Corrector. Similarly each LocalNC must be replaced for each NamespaceNC.

**Lemma 31:** The following is true for any NBCA set:

1. There is no need to collect all {*}name, {ns}* nor {ns}name nameclass to simplify the set of nameclasses: a finite set of these nameclasses will never be equal to any basic rule.
2. No nameclass from Corrector can be subset of AddBase nameclasses union. If such nameclass exists then it can be safely deleted.
3. If Base is empty then Corrector must be empty.
4. If Corrector is empty then AddBase has no meaning and must be empty.
5. If Corrector contains {*}*, then the BCA set contains qnames only from AddBase. Therefore, Base must be copied to AddBase and both Corrector and AddBase must be cleared.
6. If AddBase contains {*}* then the BCA set accepts any qname. Therefore, it must be remade such that Base contains only the {*}* nameclass, Corrector and AddBase is empty.
7. Corrector.nc is subset of Base.nc.
8. AddBase.nc is subset of Corrector.nc.
9. NBCA is empty if and only if the Base set is empty.
10. NBCA accepts any qname if and only if Base contains {*}* and Corrector is empty.

**Proof:** Part 2 is implied by the parts 7 and 8 of the NBCA definition. Part 3 is implied by the part 7 of the NBCA definition. Part 7 is equivalent to part 7 of the NBCA definition. Part 8 is equivalent to part 8 of the NBCA definition.

**Lemma 32:** If Base doesn't contain {*}*, then AddBase set is empty.
**Proof:** If Base doesn't contain {*}*, then it may contain multiple NamespaceNC, LocalNC and SimpleNC. If the Corrector contains LocalNC then Base must also contain this LocalNC otherwise rule 7 of the NBCA definition would be broken. However, this violates rule 3 of the NBCA definition hence the Corrector cannot contain LocalNC. This is similar for the NamespaceNC. Hence Corrector can contain only SimpleNC. Due to lemma 31 rule 2, AddBase cannot contain LocalNC nor NamespaceNC nor it can contain SimpleNCs that are in Corrector. Any other SimpleNC in AddBase violates definition 30 rule 6, thus AddBase set must be empty.

**Lemma 33:** Corrector never contains {*}*.
**Proof:** Use lemma 31 rule 5, then lemma 31 rule 3.

**Lemma 34:** The intersection of Corrector and basic nameclass A (except the {*}* nameclass) results in set of SimpleNCs. This set will contain A if, and only if A belongs to Corrector.
**Proof:** Corrector cannot contain {*}*. The result set is computed as an union of intersections of each nameclass in Corrector with A. Intersection with different types of basic nameclasses (except {*}*) is always SimpleNC or empty set. Second part can be proven using lemma 29. If A belongs to result set then there must exist some B in Corrector, equal to A.

**Lemma 35:** If Base contains {*}* then AddBase contains only SimpleNCs.
**Proof:** The proof is similar to the proof of lemma 32.

#### 4.3.1.1. Constructing an NBCA set from nameclass object model

We can compute this set by visiting all nameclasses as follows:

- SimpleNC, LocalNC, NamespaceNC, AnyNC – return new set with Base containing only this

nameclass, having Corrector and AddBase empty.

- NotNC – if NBCA set computed from child nameclass is empty, then result NBCA set shall have the Base set containing only {*}* and Corrector and AddBase shall be empty. If Base contains {*}* then we can simply replace Base content with Corrector contents, copy AddBase to Corrector and let the AddBase be empty. If Base does not contain {*}* and Corrector is empty then we copy Base to Corrector, let Base have only {*}* and AddBase be empty. When Base does not contain {*}*, Corrector is not empty and AddBase is empty then copy Corrector to AddBase, copy Base to Corrector and let Base be {*}*. The case, when Base does not contain {*}* and Corrector and AddBase is not empty cannot happen. No corrections are needed – the result set is NBCA.
- DifferenceNC – let's have two NBCA sets A and B made from children of this DifferenceNC. We need to compute their difference. If B accepts all qnames then return empty set. If B is empty then return A. If A is empty then return empty set. Otherwise we have to compute intersection of A and a complement of B.
- Intersection - New Base shall be computed by intersection of A.Base and BC.Base. New Corrector shall be computed by union of A.Corrector and BC.Corrector. This could break lemma 31 therefore normalization must be performed after computation. New AddBase is computed by union of two intersections: the intersection of A.AddBase and B, and the intersection of B.AddBase and A. Proof of correctiveness is beyond the scope of this work.
- ChoiceNC - let's have two NBCA sets computed from children of this ChoiceNC. We need to compute A union B. In fact, we can compute the union using complement and difference.

To get any qname that the NBCA set accepts, we have to follow these steps:

1. If AnyBase is not empty then, due to lemma 35 and lemma 32 it may contain only SimpleNC instances so simply return any SimpleNC from this list.
2. If both AnyBase and Corrector are empty then return any qname from any nameclass from Base.
3. If AnyBase is empty and Corrector is not empty then we can say that any nameclass from the Base set differenced by the Corrector must result in non-empty set. Thus we can take any nameclass from Base and find first qname that is not accepted by the Corrector.

## 4.4. Implementation of atomic operations

### 4.4.1. Element creation

We have a point in the document where the user wants to insert some element. Our task is to guide the user through the process of creating an element (or multiple elements and attributes). We shall assume that document is valid before the execution of the algorithm. Of course this applies to all following algorithms which implements an atomic operation.

#### 4.4.1.1. InsertList

The InsertList is a sequence of the objects called ElementLoc that have following content:

- The InsertPoint object, denoting the point where new element shall be inserted.
- The element rule determining the qname and rule for the newly created content.

The InsertPoint consists of the following:

- The insertpoint containing the order number of the node. The element will be inserted before this node.
- If the insertpoint denotes a text node, then the pos property may be specified - it contains the index of the character in this text node. The element shall be inserted before this character.

Two InsertPoints are equal when their two properties are equal. Two ElementLocs are equal when their InsertPoints and their element rules are equal.

The InsertList shall be used to hold all elements that are required to be created at the same time in some element (this means that we must create these elements in one atomic operation - we cannot create one element in one atomic operation and second element in another atomic operation). The sequence must be ordered: if one pair follows the second pair then the first pair InsertPoint object must be less than or equal to the second pair InsertPoint object. When the InsertPoints are equal then the insertion order depends on the order in the InsertList sequence.

The subsequenceness property has the same definition as on normal sets except that we are using the ElementLoc equality defined above.

#### 4.4.1.2. Computing InsertLists

First we will need all possible InsertLists describing all useful combinations of elements that are insertable into given element. Let's have a list L of child elements of given element. We shall start with the child rule of the element rule using the following rules:

- `<R>` - if the first item from L is an element with qname from the `<R>nc` set (if not then we will throw an error) then check whether this element complies the rule `R`. If yes then remove it from L otherwise throw an error. Throwing an error means that we picked wrong non-deterministic path and the execution will return to nearest error handler.
- `@R` - ignore this rule for now.
- `A.B` - execute the algorithm on the rule `A` and with returned list L continue the execution on the rule `B`.
- `A+` - first execute the algorithm on the rule `A`. Then establish the error handler and forever repeat the following: create a duplicate of L and execute the algorithm on the rule `A`. If an error is thrown then it will be caught in this handler. In this handler simply do nothing - last iteration was not succesful so try another rule.
- `A|B` - create an error handler and a clone of L and try rule `A` with this clone. If the execution was succesful then copy the returned list to L and return. If an error was thrown then move out from the error handler and try execution on rule `B` with the original list L.
- `A ALL B` - if we execute the algorithm on the rule `A` we must be aware that the content generated by `A` may be mixed with content generated by rule `B`. Hence we must be able to determine if

next element is generated by `B` - if yes then switch the execution to rule `B`, process the element and then switch the execution back to rule `A`. This is not possible with a recursion algorithm - we cannot switch the stacks.

It could be possible to simulate the calls in this algorithm and have control over the stack. However, this algorithm is not correct. Let's have this rule:

`<R>{E} > (<R1>{A}+).<R1>{A}.<R2>{B}`

The algorithm will eat all A elements in simulation of rule + thus the following rule `<R2>` always throws an error and we can never finish succesfully.

#### 4.4.1.2.1. The acceptor model

MSV offers an alternative way of traversing the document tree. Although it serves primarily for accepting or rejecting offered nodes, it is flexible enough to be used for various computation. The Acceptor is an object capable of eating the sequence of nodes for given rule. This is the communication scenario:

1. After its creation the acceptor expects all attributes present in the element. This step is not required - the rule may contain no direct descendant attribute rules and in such case the acceptor will reject any given attributes.
2. After all attributes had been fed the end of attributes must be announced.
3. Acceptor consumes all element, text and cdata nodes, in the order in which they are present in the document. We must concatenate the textual value of all text and cdata nodes that have no element nodes inbetween.
4. If the next content is a textual value then inform the acceptor via its `onText` method.
5. If the next content is an element then ask the acceptor to create a child acceptor which validates the content of the element. This child acceptor is also an acceptor so we can use this communication scenario again. When the child acceptor is finished then we give it to the original acceptor which finally steps over the element and is prepared to process next content.

The interesting thing is that the acceptor modifies its rule as it eats the content. When we replace the 'child nodes' term with the 'input tape' and 'rule' with 'state' then we see that the acceptor model resembles the automaton model. The actual rule of the acceptor describes the content that the acceptor expects to see as an input. We can use this feature as follows:

1. Retrieve the rule for the element where we want to insert new content.
2. Create an acceptor for the child of this rule.
3. Feed the acceptor with all attributes.
4. Feed the acceptor with text and elements until we reach the desired insert point.
5. Analyze the current acceptor rule with algorithm 36.

**Algorithm 36 (getFirstElements):** This algorithm analyzes given rule and returns a set of element rules that can stand at first place of the element sequence generated by given rule. For each rule type execute this algorithm:

- `A.B` - execute the algorithm recursively on the rule `A`. If `A` is children-ε-reducible then execute also on the rule `B`.

- `A|B` - execute the algorithm on both rules.
- `A+` - execute the algorithm on the rule `A`.
- `<R>` - add this rule to the result set.
- `A ALL B` - because the first element may be from both `A` and `B` rules then act as in the `A|B` case.
- In other cases (`@R`, ε, `AnyString` and `data`) do nothing.

The result of ε reducibility is stored directly in the MSV rule objects (these objects are called 'expressions' in MSV). The children-ε reducibility is not stored and it must be computed, hence querying the value of the ε reducibility is much quicker. We got from lemma 10 that the two properties values are equal for each rule that does not contain attribute expressions. The Acceptor state (rule) is modified this way when we announce an end of attributes. Hence we may safely use the ε reducibility in algorithm 36.

#### 4.4.1.2.2. Detection of used rule

We know that the next element must be generated by a rule from the result of algorithm 36. What if the intersection of nameclass sets of two different rules is not empty? If we let acceptor accept the qname from this intersection then we cannot tell which rule was used (let's have rule $<R_1>\{E\} \mid (<R_2>\{E\}.<R_3\{F\}) \mid <R_3>\{F\}$ and we want insert something before the already existing element `F`, we can use only rule $R_2$). We cannot compare the old and new state (after the eating of the element) of the acceptor and analyze the differences: there are rules that will not change after eating an element (for example `A+|ε`). We cannot analyze the element content because the element is currently being created - it has no content.

We can try to force the acceptor to use the rule that we want. This cannot be done by deleting other element rules - we do not know the place in the new state where to insert them back, and we must insert these rules back otherwise some possibilities may be lost. Let's try to replace them with a special element rule, having nameclass set empty, containing reference to the replaced rule. In the new state we can simply replace them with the reference.

#### 4.4.1.2.3. The rejected elements problem

Let's have a sequence of elements `A.B.C` (children of E with rule `<R>{E}`) and the following rule:

$<R_1>\{A\} . (<R_2>\{B\} \mid (<R_3>\{Q\}.<R_2>\{B\}.<R_4>\{P\})) . <R_5>\{C\}$

We want to insert some element after the B element. When the acceptor eats A and B it did not find the element Q. The choice rule in the acceptor state is replaced by single $<R_2>\{B\}$ thus we lose the possibility to insert element P. Please notice that the rules $R_3$ and $R_4$ are optional despite the fact that they may not be descendants of the ε-reducible rule.

Generally, this problem occurs when there is a Choice rule, where child 1 is capable of generating the sequence of nodes that is subsequence of the sequence of nodes generated by child 2. Let's call this choice rule the expansion rule, child 2 the stronger rule and child 1 the weaker rule. This expansion rule $R_e$ must be ascendant of rule of the element after which we are trying to insert

an element. Please note that this rule must be a direct descendant of `<R>{E}` because we have forbidden the element rule change in chapter 4.1.4: The rule identification.

Let's have a set with all possible expansion rules. We need to check each thoroughly if it is a real expansion rule. First, we must detect which elements this rule really generates. This could be quite tricky thanks to the `ALL` rule.

**Algorithm 37 (getGeneratedElements):** Let's create a new rule `R'` from `R` having all direct descendant element rules of a possible expansion point selected (we can select the rule for example by changing the nameclass namespace to a special select namespace). The child of each selected rule is equivalent to the child of the original rule.

We will create an acceptor with starting rule `R'` and start walking through the child elements. Let the next element be A. If A' (A selected) can be generated by one of the selected rules and it may have a rule amongst rules returned by algorithm 36 (otherwise we simply let the acceptor move further by consuming the A element) then we will first try to give A' to the acceptor. When the algorithm backtracks to this location then we will try to continue with the original element name. We can return by remembering older acceptor states. When A nor A' is not accepted by the acceptor then we must backtrack and try different possibility.

When the acceptor eats last element then make a list of positions of all selected elements and continue with the algorithm: we need all possible selections.

When we have a list of elements that the weaker rule generates we can start detecting if the stronger rule really is stronger. We will simply try to extend this list of elements with some new elements and check if the stronger rule accepts this new sequence. If these new elements had been found then we may insert them into E. They must be inserted in the same order in which they appear in the extended list, and between the same elements.

**Algorithm 38 (verifyStrongerRule):** Create an acceptor for the stronger rule. We will split the algorithm into three phases, depending on current location in the element list:

- First phase: the acceptor have not ate the B element yet. We are trying to explore every possibility in order to have each possible element insertable after the B element. Hence we first try to give the acceptor a new, created element instead of the next element from the children list of E (we pick the element names from algorithm 36). When all creatable elements have been checked, we can move to consume the next element.
- Second phase: the last element that was eaten by the acceptor was element B. We must insert some new element hence we must skip the cases when there is no new element insertable.
- Third phase: we are trying to check as quickly as possible if the acceptor can accept the element sequence therefore we let the acceptor eat next element whenever possible.

Let's have the rule set from algorithm 36. We have to perform the following before consuming/creating next node:

- If the rule set does not accept the next element in the element sequence then we must attempt to create new element and give it to the acceptor. This element must be from this rule set. If no attempt was succesfull then we must backtrack to a previous step. If we cannot backtrack then

fail: the rule is not a stronger rule.

- Rule set accepts the next element. If we are presently in first or second phase then try to give a newly created element to the acceptor first. In third phase try to give the acceptor the next element first. When creating an element we must take care not to create them indefinitely (when rule contains the + rule). This problem does not occur when walking over a next element - the element list has a finite length. The + rule may be applied only once in the element creation sequence - if we cannot move forward in the element list after one application of the + rule then we cannot move forward after any number of application. To prevent infinite cycling we may compute the maximal length of element sequence that we can consecutively create. This ensures that the algorithm will always finish.
- When we accepted the last element from the element list then we do not have to test if the acceptor accepts: we may generate the required content from the acceptor state.

We must compute possible qnames of elements that we shall try to create. For each rule present in the result of algorithm 36 only one qname must be picked in order to maximize the speed of the algorithm. This qname should not be accepted by other rules nameclasses in order to minimise the need of usage of the algorithm introduced in chapter 4.4.1.2.2: Detection of used rule. Hence we must compute the system of representants on the set of these nameclasses sets. If we are unable to compute representant for some set then we must choose a minimal set of colliding qnames - they must be processed by acceptor in a special way, as described in chapter 4.4.1.2.2: Detection of used rule.

**Algorithm 39 (getMaxElementsCount):** We can compute the maximal length of creatable element sequence with this recursive algorithm. For each rule type do the following:

- `<R>` - return 1.
- `MIX(R)`, `R+` - compute the result for the child rule.
- ε, `@R`, `data`, `AnyString` - return 0.
- `A.B`, `A ALL B` - return the sum of results for A and B.
- `A|B` - execute the algorithm for both children and return the greater number.

#### 4.4.1.2.4. Optimalisation

Let's have an InsertList set from the previously described algorithm. When we insert any one InsertList from the set into E we get a valid content. However this set is too large - it includes all possibilities. Let's try to find a way how to delete unnecessary InsertLists. We may safely delete an InsertList that can be replaced by consecutive insertion of simplier InsertLists.

**Lemma 40:** Let $L_1$ and $L_2$ be two different InsertLists. If $L_1$ is subsequence of $L_2$ then we may safely delete the $L_2$ InsertList from the result set.

**Proof:** We can create E extended by $L_2$ using this algorithm:

1. Extend E by $L_1$. This extension will result in a valid content, thus we may use the element creation algorithm on this extension.
2. Let's take the first ElementLoc `el` from $L_2$ that is not in $L_1$. The algorithm will find all

possibilities for inserting an element hence it must return at least one InsertList $L_{el}$ containing `el`.

3. Let's choose such $L_{el}$ that is subsequence of $L_2$. This InsertList must exist - the algorithm will find all valid possibilities and at least one possiblity must lead to finding $L_2$, otherwise we could not find $L_2$.
4. Extend E by $L_{el}$ and let $L_1$ be $L_1$ union $L_{el}$. If $L_1$ is not equal to $L_2$ then repeat this algorithm from step 2.

Hence we can compute E extended by $L_2$ using $L_1$ and other InsertLists thus we can remove $L_2$ from the result set.

We have a simple rule to reduce the number of result InsertLists. How can we implement this rule into algorithm 38 in order to find out for the InsertList being created if it is a supersequence of some InsertList that is already present in the result set? The created InsertList could be compared after each modification with the result set. However, to verify that $L_1$ is a subsequence of $L_2$ we must check whole $L_1$ and at least [length of $L_1$] elements of $L_2$. $L_2$ is being created by the algorithm hence we shall try to minimise the length of scanned part of $L_2$ in order for the algorithm to detect such InsertList sooner.

The length of $L_1$ (the InsertList already found) can be minimalised by trying to find shortest sequences first. An ideal InsertList is an InsertList with exactly one item: the item that can be inserted after given element. If we try to generate elements first in the first phase of algorithm 38 then we will only lengthen the InsertList being created. Hence we shall modify the first phase definition: the algorithm must try to walk over the next element first. This assures that shorter InsertLists will be placed sooner to the result set. This even assures that the InsertList being created will never be a subsequence of some InsertLists already placed in the result set.

To avoid constant subsequence tests we shall remember the acceptor states also. Let's have a set S containing the following triples:

- state - the state rule of the acceptor.
- InsertPoint - the insertpoint where the acceptor is standing.
- InsertList - the InsertList being created.

This triple denotes a 'state' of the algorithm. Every triple placed in the S set says that the algorithm already visited this state hence there is no need to continue further and we must backtrack. Let the InsertList being created be $I_c$ and let S contain the following triple:

- state is equal to current state of the acceptor.
- InsertPoint is equal to actual insertpoint.
- InsertList is a subsequence of $I_c$.

If such triple exists then we will not generate new element sequence because we already tried to generate all sequences with this acceptor state from this point. If we find any sequence from this point we know that we already found this sequence earlier. Hence if $I_c$ is a supersequence of the InsertList then it will stay a supersequence of some already checked sequence, thus we can safely backtrack and check another possibilities. This approach even shields us from the infinite creation

of elements hence we have no need to use algorithm 39 anymore.

Let's have rule

```
(<RB>{B}.<RD>{D}) | (<RA>{A}.<RB>{B}.<RD>{D}) | (<RA>{A}.<RB>{B}.<RC>{C}.<RD>{D})
```

and let the content of E be only B. Let's try to insert something after B. The algorithm first walks over B with remaining rule `<RD>`. Later when it inserts A before B, steps over B and generates C, it finds itself with the same remaining expression `<RD>`, thus discarding this state as a state in which it had been before. That means that we lose the possibility of inserting the C element. In order to prevent this situation the algorithm must step further when the new possibility of inserting an element after B had been found even when this state is already in the set of states. This can be achieved if we deny insertion of the state that has empty InsertList and InsertPoint pointing right after the B element. This ensures that:

- The acceptor must generate new element when it is right after the B element
- When the acceptor does so and it finds new element then the current InsertList can not be a supersequence of an existing InsertList.

#### 4.4.1.2.5. Text

In algorithm 38 we ignored the presence of text and cdata nodes between element nodes. Let's see how the algorithm must be modified. Please look at the following rule:

```
(<RA>{A}+) . (MIX(<RB>{B}+))
```

Let E have the following content: `<A/><A/>text<B/>text<B/>text<B/>text`. We can insert both A and B after the second A element but we cannot insert A into the first text. In order to check all possibilities we should try three places instead of one: before the text, into the text and after the text. This should triple the count of insert places hence an optimalisation is required.

From axiom 24 and the fact that `AnyString` accepts any string we got that if some rule accepts `text1<E>text2` then it must accept also both `<E>text1text2` and `text1text2<E>`. Hence all elements able to stand in some text are able to stand before or after this text thus we do not need to examine the second position. However the fact that element can stand in both places does not imply that it can stand inbetween the text, hence if we are required to insert the element into the text then we must examine also this place. However this is needed only in this place.

#### 4.4.1.2.6. Inserting into descendant elements

Please notice that algorithm 38 does not try to insert new elements into descendants of children element of E. This case can happen - take a look on the following grammar:

```
<Rroot>{root} > <RA1>{A} | (<RA2>{A}.<RB>{B})
<RA1>{A} > <RC>{C}
<RA2>{A} > <RC>{C}.<RD>{D}
```

Let the root element contain one A element, A containing one C element. root can be expanded by inserting B after the A element. This modifies the rule generating element A thus we must insert D after C. This expansion is undescribable by current InsertList model. However we disallowed the change of the element rule in chapter 4.1.4: The rule identification hence we are

shielded from such cases.

#### 4.4.1.2.7. Conclusion

We described the brute-force algorithm for inserting elements. There are simplier algorithms for some specific rules however the task to construct the algorithm had been completed. Please notice that slightly modified algorithm 38 can be used also on the whole content of E using the child rule of the E rule instead of the stronger rule. These modifications are:

- If we cannot backtrack (we checked all possibilities) then finish succesfully and return the set of InsertLists found.
- When we accepted last element then put current InsertList to a result set and try other possibilities.

The algorithm is used this way in the current implementation because using it on the weaker/stronger rules brings some complications. First, all InsertPoints in computed InsertLists must be adjusted to be valid in the context of the original element. Moreover, algorithm 37 is not capable of selecting the text elements hence we are unable to tell which text nodes were generated by the weaker rule. The absence of text nodes can bring some unwanted results.

### 4.4.1.3. Filling the content of created elements

Let's have a new empty element and a rule for this element. Our task is to create all required elements, attributes and text in order for the element to be valid. First we build a list of all required attributes. Existence of an attribute does not set conditions on existence of any other attribute hence we do not have to pick between attributes thus it is really a list of attributes. User must type a valid textual value for each required attribute.

When attributes are created we can start with creating elements or text. If a textual value is required then elements are prohibited hence we just ask the user for this textual value. If elements will be present in the element then no textual value is required thus no text entering should be allowed - we are creating only the required content. To dispose the rule from the attribute rules we simply create an acceptor and feed it with the created attributes.

To create all required elements the following algorithm may be used: do a inorder traverse over the tree of rules and if the choice occurs (not ε-reducible) then let the user choose between the two rules. The disadvantage of such approach is that we cannot visualise the rule properly thus the user does not see properly the outcomes of each choice. Hence we shall generate a list of InsertLists, each InsertList will represent one possibility of element sequence. Because no elements are currently present in the element, each ElementLoc will have zero InsertPoint.

**Algorithm 41 (createElementContent):** Let's have the rule `R`. We must compute list of InsertLists, each a valid combination of required elements. We shall use this recursive algorithm, starting with `R`:

- `AnyString`, `data`, ε - return an empty set.
- `@R` - cannot occur, the expression must not contain attribute rules.

- `<R>` - return set containing one InsertList, made from zero insertpoint and this rule.
- `MIX(R), R+` - if `R` is not ε-reducible then return result of execute the algorithm on `R`, otherwise return an empty set.
- `A|B` - if the rule is ε-reducible then return empty set. Otherwise return union of sets returned by the execution on both `A` and `B`.
- `A ALL B, A.B` - we can treat the `ALL` as sequence - elements can be moved later. We need to compute sets for both `A` and `B` and combine them together: each one InsertList from `A` must be chained with each one InsertList from `B`. All these chained InsertLists are returned as a set.

User must choose the InsertList from returned list. For each ElementLoc the element must be created at appropriate position and this algorithm must be repeated for this element. This applies also to a very first InsertList, returned by the algorithm in chapter 4.4.1.2: Computing InsertLists.

## 4.4.2. Element deletion

The simpliest algorithm for the element deletion is as follows: let's have element E, its rule $R_E$. We want to delete some child elements. We create an acceptor for child rule of $R_E$ and we shall try to validate the sequence of children of E without the elements we want to delete. If the acceptor accepts this sequence then these elements can be safely deleted.

Let's suppose that acceptor stops before element P (when it tries to consume P it throws an error). Thus we know that P will be accepted by no rule from the set returned by algorithm 36, executed on acceptor current state. Here we must decide whether we shall try to create a new element and insert it before P, or we try to delete P and continue. We cannot blindly prefer one of these options: we can end with generating many elements, or with deleting too many elements. A good heuristic may be to analyse the rule. For example we are about to delete an optional element A, having rule $R_A$. Let the rule somewhere in $R_E$ be $R_A$ `|` $R_B${B}. We know that the simplest way is to replace B with A. On the other hand if the rule is $R_A$ `.` $R_B${B} then it is better to delete B.

The rule may be quite complicated so the analysis can be quite hard. If schema has ambigous rules then we may be unable to find the correct rule derivation of the element. Moreover, `ALL` alters the sibling consistency: let $R_1$ generate the sequence $P_1.P_2$ and $R_2$ the sequence $Q_1.Q_2$. $P_1$ is a sibling with $P_2$, and $Q_1$ is a sibling with $Q_2$. However, $R_1$ `ALL` $R_2$ is capable of generating $P_1.Q_1.P_2.Q_2$. Due to these complications the issue is open.

The current implementation will always delete an element if it is not accepted by the acceptor. If the acceptor will not accept the whole sequence then the parent element must be deleted, hence we shall execute the algorithm on the parent element.

## 4.4.3. Element movement

The `ALL` rule allows us to create a rule that is able to accept elements in any order. For example $<R_A>${A} `ALL (`$<R_B>${B} `ALL` $<R_C>${C}`)` accept any sequence of elements A, B, C, however all three must be present. Hence we cannot use two atomic operations: delete the element and create it somewhere else, thus we shall need the ability to move the element.

Suppose that we want to move an element E, contained in nametree $N_E$. Let the parent of E be $E_P$ and its nametree $N_P$. If E is the nameroot of $N_E$ (hence $N_P$ is a different nametree than $N_E$) then rules for E and $E_P$ belongs to different grammars. We can say the following:

- If E is not one of the possible nameroots then we cannot move it into a nametree with different namespace.
- If $N_P$ is not valid when E is removed then we can move E only in scope of $N_P$. This property could be verifiable only by validating whole nametree $N_P$ if we would permit the rule change. However this is forbidden in chapter 4.1.4: The rule identification hence it is sufficient to test only the content of $E_P$.
- We cannot move E into itself. This implies that the root element is unmovable.

Let $R_E$ be a element rule for an element E. We must examine each grammar for each nametree in which we would like to insert E. Let $R_{EP}$ be set of element rules that are direct ascendants of $R_E$ - these rules are rules for such elements, into which we can move the E element. For each rule the set of elements that are generated by this rule must be identified.

For each element in the set we must test whether E can be inserted among its children. Let this element be $E_I$ and its rule $R_I$. We construct the acceptor from the child of $R_I$ and let it eat the attributes. We shall walk with this acceptor over the children of $E_I$. Before each element we try to insert the E element and then try to walk to the end of the element sequence. If the acceptor accepts then the position of E is valid and E can be moved here. If $E_I$ is $E_P$ then we must ignore the E still present in $E_P$ and we must not try to insert E into its original position.

We must verify the points in the text also. However if the element is insertable into one position of the text/cdata node N then it is insertable into any sibling text/cdata node, not separated from N by an element. Hence only one pointer is required to identify such places.

To optimise this algorithm we can once again remember the algorithm states. For each (InsertPoint, acceptor state) pair visited we remember if this state leads to acception or rejection. We remember and check the pairs only when walking to the end of the children sequence.

## 4.4.4. Enclosing and declosing nodes

The enclosing of given list of adjacent nodes means that we create new element, insert it before the first node and move all nodes into this element. Hence we are searching for all rules that:

- Generates an element that can be inserted at the place of the first node,
- This element can contain these nodes as children, and
- The element parent is valid when the nodes are replaced by this element.

Let the parent of all these nodes be $E_P$. We create an acceptor and let it eat all nodes until the first given node has been reached. Here we analyze the acceptor state with algorithm 36. Each element rule must be tested if it is capable of generating an element containing the nodes. In order to test the rule all direct descendant attribute rules must be replaced by ε. We can now simply create the acceptor from this modified rule, immediately announce the end of attributes and validate the given nodes with the acceptor. For each rule that passed the test compute the system of

representants. Each qname must be given to the acceptor, then the input nodes must be skipped and the rest of the children must be validated. If the validation succeeds then the rule may be used to generate the element that encloses given nodes.

When user chooses the element and fills the values of required attributes then the enclosing can be finally performed. The list of required attribute rules can be computed by the following algorithm.

**Algorithm 42 (computeRequiredAttributes):** Let $R_E$ be rule for element E. We must compute the set of attribute expressions required to occur in E. We shall compute the set recursively on the child of $R_E$.

- If the rule is ε-reducible then do nothing - it cannot contain required attribute rules.
- If the rule is an element rule then do nothing - we are searching for direct descendant attribute rules only.
- If the rule is an attribute rule then add it to the result set.
- Otherwise we recursively execute the algorithm on all children of the rule.

The declosing process is an inverse operation to the enclosing operation - selected element must be replaced by its children. We simply validate the parent element, pretending that the element is already replaced by its children. If the validation succeeds then the declosing operation can be performed.

### 4.4.5. Attribute creation

We must compute the set of all attribute expressions that can be created in given element E. If the attribute is already present in the element then we cannot create another attribute with the rule. The rule + is not allowed to have a direct descendant attribute. Only the @EP attribute rules may be enclosed in +.

We do not have to take children elements/text/cdata nodes of E into account nor do we have to analyze the | rules specially (to determine if it is a choice between two attribute rules for example) - this is guaranteed by lemma 22. We can use these features in the following recursive algorithm:

- If the rule is an element rule then do nothing.
- If the rule is an attribute then check if it accepts an attribute present in E. If yes then mark the attribute as matched - we must not match this attribute again to some rule. If not then add it to the result set.
- Otherwise execute the algorithm on all children of the rule.

However the algorithm will not work when one rule may generate two attributes - we do not know which attribute to mark as matched. Hence we shall use the acceptor again. Attributes does not have any particular order therefore they can be consumed by acceptor in any order.

**Algorithm 43 (creatableAttributesList):** We have element E and its rule $R_E$. We must compute the set of attribute rules that can be created in E. We start with creating an acceptor for the child rule of $R_E$. Let the acceptor consume all attributes present in the element. Now analyze the state and return

all direct descendant attribute rules. We do not have to check further the required attributes - no attribute existence is contingent on existence of another attribute.

### 4.4.6. Attribute textual value modification

This algorithm could be simple: we find the attribute rule for given attribute A and retrieve the set of possible `data` rules. Child of the attribute rule cannot contain `+`, `ALL`, `.` nor `MIX` rules - no concatenation of two or more `data` rules is allowed. The only rule that can be descendants of attribute expression are the `data` and `|` rules hence all `data` rules are alternatives to each other thus the textual value must be accepted by at least one of these DataExps. Unfortunately in some cases we cannot get correct expression (when two rules are able to generate an attribute with same qname and textual value) so we must once again use an acceptor.

Let the acceptor eat all attributes except the one we are willing to modify. Then check the acceptor state. Each direct descendant attribute rule that have the qname of A in NameClass set is potentially capable of generating our attribute. According to lemma 22 we do not have to create other attributes thus we can use all these attribute rules, such that their nameclass sets accept the qname of the attribute. They are alternatives to each other so the new textual value must be accepted by at least one `data` rule present in any of these attribute rules.

### 4.4.7. Attribute deletion

Because we may be unable to identify the attribute rule which generates given attribute, we can’t check whether its rule is optional or not. We shall use the acceptor and base the algorithm on the fact that attributes can be consumed in any order. Feed all attributes present in the element except the one we are querying. If we can’t properly announce the end of attributes then this attribute is required and thus cannot be removed.

### 4.4.8. Text creation

Let's have element E and InsertPoint I specifying place, where we want to insert a new text. The user may specify the type of the node that shall be created: it may be text or cdata node type. The type may be defaulted to a text node. If the InsertPoint points next to a node with same type then we must modify this node instead - there must be no sibling text or cdata nodes in the document.

If E already contains some elements then the text present in E can be generated using the `AnyString` rule only. Thus, we shall try to accept the sequence of children of E extended with a 'virtual' node inserted at I. This virtual node must have any non-empty text value. If the validation succeeds then any string can be inserted at I. This algorithm may be optimized: if there is at least one text/cdata sibling node not separated from node denoted by I by an element (it may be separated for example by a comment nodes) then we can create any text.

If E contains no elements then we must detect all direct descendant `data` rules that may be

able to generate the textual content. We are searching for the rules for the textual value of E hence these rules must not be descendants of an attribute rule, that is a descendant of $R_E$. Moreover, no `data` rule must require the presence of an element. These `data` rules can only be alternatives to each other as stated in chapter 4.4.6: Attribute textual value modification.

**Algorithm 44 (getTextRule):** We need to detect the set of alternative `data` rules that may generate textual value of the element. We shall execute the following recursive algorithm on the child of $R_E$. Each step returns a pair: first component is true when some elements need to be created, second component is a set of `data` rules presently found.

- `<R>` - return `(true, {})`.
- `@R` - return `(false, {})`.
- `A|B` - if both A and B returns `(true, {})` then return `(true, {})` otherwise return `true` and the union of both sets.
- `A ALL B`, `A.B` - if at least one child returns `(true, {})` then return `(true, {})` otherwise return `true` and the union of both sets. If both children returned `true` and non-empty sets then signalise an error - datatypes cannot be concatenated nor interleaved.
- `MIX(A)` - if A returns `(true, {})` then return `(true, {})` otherwise return `(false, {AnyString})`. If A returned `false` and a non-empty set then signalise an error.
- `A+` - return the result of A.
- `data` - return the `(false, {data})` pair.
- ε - return `(false, {})`.
- `AnyString` - return `(false, {AnyString})`.

If the result set contains the `AnyString` rule then all other rules may be removed from the set.

A special care must be taken when creating a text node using a rule different from the `AnyString` rule. There may be multiple text/cdata nodes and they may even be separated by a comment or processing instruction nodes (there cannot be elements if the rule is not an `AnyString` rule). Hence the values of all child text/cdata nodes must be concatenated and this result must be validated against the ruleset.

## 4.4.9. Text modification and deletion

If E contains an element then all text/cdata nodes are accepted by the `AnyString` rule hence they can be modified in any way. Otherwise we shall use algorithm 44 and check if the new value is valid. If the new text value is an empty string then the text/cdata node must be removed instead.

If E contains an element then any text is always removable. Let E contain no element. If E contains more than one text/cdata node then we are modifying the text value of E instead. Let E contain only one text/cdata node that we are about to delete. We could simply query the child of $R_E$ if it is children-ε-reducible. However, for some unknown reason the `AnyString` in the MSV implementation is not ε-reducible hence it is not children-ε-reducible. Thus we must get the ruleset using algorithm 44 and check if it accepts an empty string.

# Conclusion

# 5

We have presented the overall WYSIWYG XML editor design. The editor uses freely available, standard technologies hence it may be freely distributed itself. We have described the core of the editor engine - the transformation chain. Most important outcome of this thesis is the description of algorithms, which allow us to build a universal semi-WYSIWYG XML editor module. Full WYSIWYG capability cannot be achieved by this editor module because the rendering outputs may vary greatly. However the editor module may be used as a base module for creating more specific editors, intended to work with some namespace only. These editors have a high potential being fully WYSIWYG.

Furthermore, we described the technique of validating the document with multiple namespaces and to use multiple schema together. This specification is an important step towards the multi-namespaceness because the schema cooperation is not yet well defined.

Some of described algorithms are not optimal: they are a brute-force algorithms, although quite optimalized. Much faster algorithms may be developed and used, however they may require a complicated schema modification. However the task to prove that the XML Document is able to be edited in a WYSIWYG fashion had been completed. The diploma thesis itself is stored in an XML document, having its own namespace (the "http://www.uniba.sk/euromath2/2004/diploma-thesis" namespace), schema and transformer to a XSL-FO language.

The EuroMath2 editor can be found at the following URL: "http://www.dcs.fmph.uniba.sk/sioux".

# Appendices

# XSLT Modification Hints

A

Here are some hints illustrating how XSLT scripts must be modified to be able to transfer special attributes and elements correctly. First of all, the EuroMath2 namespace must be defined by putting the following attribute to a root element of XSLT:

```
xmlns:sioux="http://www.uniba.sk/euromath2/reserved"
```

The following template must be used to handle em2p:mark elements. High priority should be set to ensure that it is always used:

```
<xsl:template match="em2p:mark">
  <xsl:copy-of select="self::*"/>
</xsl:template>
```

However, you may safely ignore processing these elements if schema doesn't support inserting elements from another namespaces. This template prints out the ID of nodeset:

```
<!-- Prints ID of given node. -->
<xsl:template match="@*" mode="em2p.id" priority="1">
 <xsl:value-of select="../@em2p:id"/><xsl:text>@</xsl:text><xsl:value-of
select="name()"/>
</xsl:template>
<xsl:template match="*" mode="em2p.id" priority="1">
 <xsl:value-of select="@em2p:id"/>
</xsl:template>
<xsl:template match="node()" mode="em2p.id" priority="0">
 <xsl:value-of select="../@em2p:id"/><xsl:text>;</xsl:text><xsl:value-of
select="count(preceding-sibling::node())"/>
</xsl:template>
```

To instantiate attribute containing ID of given node, the following snippet can be used: (note that this attribute is instantiated only when the em2p:id attribute exists, thus the XSLT is usable with as well as without EuroMath2)

```
<!-- Instantiates an id attribute for given node. -->
<xsl:template match="*" mode="em2p.createIdAttr" priority="1">
 <xsl:if test="@em2p:id">
  <xsl:attribute name="id"><xsl:apply-templates select="."
mode="em2p.id"/></xsl:attribute>
 </xsl:if>
</xsl:template>
<xsl:template match="node()" mode="em2p.createIdAttr" priority="0">
 <xsl:if test="../@em2p:id">
  <xsl:attribute name="id"><xsl:apply-templates select="."
```

```
mode="em2p.id"/></xsl:attribute>
 </xsl:if>
</xsl:template>
```

In order to transport text and cdata IDs correctly, this XSLT command `<xsl:value-of select="nodeset">` must not be used. This command `<xsl:apply-templates select="nodeset" mode="em2p.text">` must be used instead, and this template must be present in XSLT script:

```
<!-- Prints text value of given node with proper IDs, as if by the
<xsl:value-of/> call. -->
<xsl:template match="node()" mode="em2p.text">
 <xsl:choose>
  <xsl:when test="self::*">
   <xsl:apply-templates mode="em2p.text"/>
  </xsl:when>
  <xsl:when test="self::text()">
   <fo:inline>
    <xsl:apply-templates select="." mode="em2p.createIdAttr">
    <xsl:value-of select="."/>
   </fo:inline>
  </xsl:when>
 </xsl:choose>
</xsl:template>
```

# References

[XML98] W3 Community: *XML Core Specification*, 1998. http://www.w3.org/XML/Core/#Publications.

[XNS99] W3 Community: *Namespaces in XML*, 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[MSV04] Kohsuke Kawaguchi: *Sun Multi-Schema XML Validator*, 2004. http://wwws.sun.com/software/xml/developers/multischema.

[XSLT03] W3 Community: *XSL Transformations*, 2003. http://www.w3.org/TR/xslt20.

[XSLFO01] W3 Community: *Extensible Stylesheet Language*, 2001. http://www.w3.org/TR/xsl.

[SED03] Andrej Adamovský: *Štandardy elektronických dokumentov*, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, 2003.

[RNG01] James Clark, Murata Makoto: *Relax NG validation schema*, 2001. http://www.relaxng.org/.

[TREX] James Clark: *Tree Regular Expressions for XML*, . http://www.thaiopensource.com/trex.

[XSD01] W3 Community: *XML Schema validation schema*, 2001. http://www.w3.org/XML/Schema.

[DOM3] W3 Community: *Document Object Model*, Level 3. http://www.w3.org/DOM/.

[TRaX] The Apache XML Project: *Transformation API for XML*, 1.0. http://xml.apache.org/xalan-j/trax.html.

[XER262] The Apache XML Project: *Xerces DOM implementation*, 2.6.2. http://xml.apache.org/xerces2-j.

[XAL260] The Apache XML Project: *Xalan-Java XSLT processor*, 2.6.0. http://xml.apache.org/xalan-j.

[FOP0205] The Apache XML Project: *Formatting Objects Processor*, 0.20.5. http://xml.apache.org/fop/.