Narrowing down XML Template Expansion and Schema Validation

René Haberland
Technical University of Dresden, Free State of Saxony, Germany
Saint Petersburg State University, Saint Petersburg, Russia
(translation into English from 2007)

Keywords. ***XML, XML-template, XML-schema, language unification, XSLT, DTD, XSD, RelaxNG, PHP, template language, template engine, tree automaton, document instantiation.***

I. Introduction

This section gives an overview of template expansion and schema validation. In the beginning, a motivational example is given to introduce the topic of this thesis. Existing contributions are presented.

A. Motivation

In XSLT, an application developer is often confronted with two fundamental problems related to XML documents: First, an XSLT-*style sheet* needs to be instantiated. Second, a given XML document, one part of a more complex XML processing pipeline, needs to be validated against a known XML schema. The automatic generation of XML documents can be very diverse and may include numerous and even distributed applications, and so is the validation of such. For example, the instantiation of the following XSLT-stylesheet fragment:

```
<xsl:template match="/">
  <foos>
    <xsl:value-of select="//page"/>
    <bar/>
  </foos>
</xsl:template>
```

may be transformed into the next document by corresponding queries to the source document:

```
<foos>
  1<bar/>
</foos>
```

In order to check validity, a RelaxNG schema may be required, for instance:

```
<element name="foos">
  <group>
    <data type="int"/>
    <element name="bar">
      <empty/>
    </element>
  </group>
</element>
```

Hence, apart from an XSLT-stylesheet an XML-schema is required a posteriori, which on the first view does not look similar to the stylesheet. If only a schema could automatically be derived from a given stylesheet, both processes could be simplified. Even better, a unified view could serve this. Saying this, the total efforts saved is vital, significantly when dropping document meta-information of approximately half with quite a diverse document structure.

This approach is tracked by the minimalistic template-language XTL [32] used for XML documents. Within this work, an instantiator and validator for XTL are implemented. Apart from that, unification, in general, is investigated. Mainly, the following issues are discussed:

- **Formalisation of instantiation and validation.**
  How can both semantics be formalised?

  Which commonalities and differences do both semantics have?
- **Demonstration of examples.**
  How do prototypical implementations look?

  Which other properties may be revealed?
- **Requirements for unification.**
  Does unification disproportionately restrict either template expansion or schema validation? – How can such restrictions be overcome, if any.

  Which restrictions are tolerable and which are not?
- **Comparison of instantiation and validation.**
  Why does a comparison always have to focus on schema validation first?

  What are reasonable criteria for comparison? Which schema languages are suitable for comparison?

  What are the limitations of XTL, and what would be reasonable extensions?

  What about usability towards XTL?

B. Preparations

This section introduces basics and related work as well as bordering disciplines.

1) Existing work: **RelaxNG-validator.**

[17] proposes the algorithm which is used currently in validating RelaxNG-documents. For instance, based on Clark's

approach, Torben Kuseler [44] developed the Haskell-XML-Toolbox [58].

Transformation of regular expressions into an automaton. [63] provides practical instructions for the construction in Haskell for a string recognising determined automaton.

[9] presents the construction of so-called *Glushkov*-automata, which are stepwise built up by deriving transitions from previously set states.

The paper [4] widens the definitions of a partial derivation according to mathematical analysis and proposes a transition calculation parametrised in comparison to [9].

Tree Automata. In [50] and [49], Makoto Murata introduces tree automata's syntax applied to XML documents. Essential terms, particularly tree grammars and languages, as well as their properties, are provided.

[16] discusses the transformation of schema documents into regular tree expressions. The construction of tree automata is discussed in the context of database applications.

[12] represents a compendium of techniques on tree automata and their applications.

Comparisons and Field Studies. Murata, Lee and Mani [48] introduce a classification of XML-schema languages. In [46], Lee and Chu classify expressibility for selected schema languages.

Rahm and Bernstein [57] propose a classification of schema-matchers.

Others. Both [2] and [61] provide a survey on denotational semantics. [62] gives an introduction to Haskell. Additional material on Haskell and functional programming may be found in [55] and [56].

2) Related Work: template engines are introduced in [53] and [52], where moreover [52] also provides a closer look from a Model-View-Controller perspective. Evaluation improvements are discussed in [43] on template expansion and in [8] on schema validation. [1] deals with XML typing in general, and particularly with *type isomorphism* in Haskell. The overall meaning of polymorphism in terms of XML is discussed in [36].

Lazy evaluation of XML-documents, refactorings of functional programs [21], [45], [64], [13] and monads/arrows [39], [37], [25], [34], [38], [66] are closely related to Haskell. Particularly, [43], [51] deals with lazy parsing of XML-documents, and [6] deals with XML-document updates.

3) Cross-bordering disciplines: Besides the just mentioned topics, this work's topic (cross-)borders with further disciplines:

- document and schema transformation,
- parsing,
- functional programming and
- XML data binding.

C. Structure of this work

Section 2 introduces basics, e.g. instantiation and validation. Section 3 investigates instantiation and validation as function. New requirements for programs to be developed and to a unification process are formulated.

Semantics for both instantiation and validation of XTL is defined in section 4. Properties are discussed. The desired data model is introduced which is to simplify semantics. Section 5 presents the software implementation of the previously defined semantics and data models in Haskell. A design for an object-oriented implementation to be continued is discussed. According to previously defined criteria, section 6 compares template expansion and schema validation in general and specifically for a selected set of XML schema languages. The focus is the unification of both. Special attention is paid to the semantics and syntax of schema languages. Complexity is considered, primarily, w.r.t. practical needs.

"'XML-Schema"' denotes W3C's proposed XML-schema language, which is often better known as "'XSD"'.

All terms are described in the glossary attached and in the following text before its first occurrence.

If a term appears in italics for the second time, this indicates its glossary meaning may deviate depending on its context. Used acronyms appear in italics on their first occurrence in the text. References to the appendices are linked in the text. Only stereotypes/patterns in sections sect.III, V may appear in italics and do not require further explanation, which is following roles as proposed by [27] and [42].

At the beginning of each section, a short overview and methodology are provided. Questions raised are either answered immediately or within the following section(s) and are indented. References, except mentioned differently in the text, are always related to prior statements.

II. Basics

This section introduces instantiation and validation, defines basic terms with illustrative examples, and introduces the template-language XTL and theoretical foundations. The modelling of XML documents is described based on a tree grammar as well as the translation.

A. Instantiation

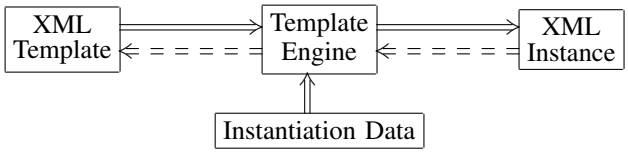


Fig. 1. Instantiation and Validation

The *instantiation* of XML documents is when a *template engine* generates an *instance* of a *template* based on some *source language* with some *instantiation data*. The result is called an *instance* and follows a *target language*'s constraints (see fig.1). At this moment, it is agreed templates are present in XML. It is further agreed on instances have to be in well-formed XML format.

A template consists of an arbitrary number of *tags* and *slots* [53], which also may be nested. In general, *slots* have to be unique among all other nodes. Slots bind instantiation data, which may differ in general. Instantiation data maybe XML or form a relation. A slot may at most refer to one source. Within an instantiation step, common tags are copied to the instance document, where the template engine evaluates a slot. Due to its tree structure, XML documents are usually processed top-down. So, the instantiation of templates is processed in pre-order (cf. sect.II-D1). W.l.o.g. it is agreed that during an instantiation step, access to already instantiated nodes is prohibited due to the transparency of the chosen model (see sect.II-D).

Slots are interfaces. Its purpose is to load data from external repositories into the instance. The loading is controlled by queries that come from the instantiation data. Queries are part of the slots. Neither its syntax nor its semantic have to follow XML or any a priori rules whatsoever. For example, a slot may contain queries in *XPath* or *SQL*. Some source languages allow more precise queries than XPath does. For instance, the returned value has to be formatted. Settings, e.g. for XTL, will be interpreted by *Placeholder-Plugins* (*PHPs*; see sect.II-C). That is why every template (e.g. a PHP website) has to follow previously agreed interfaces that control instantiation data access.

For distinguishing a document instantiation from, for instance, object instantiation, the term *template expansion* is sometimes used [53],[52]. The term "'expansion"' tries to illustrate something is replaced by something more extensive. However, the replacement does not always have to be longer. The opposite may be true. A slot is replaced by an arbitrary number of nodes. The replacement can be interpreted as reduction rather than expansion if the node(s) to be inserted are in total shorter than the slot. Hence, it is agreed that both instantiation and template expansion may be used as synonyms.

For demonstration purpose, the instantiation of the following XSLT-stylesheet with XPath [11] as embedded query language is shown:

```
<?xml version="1.0" encoding="UTF-8"?>                    1
<xsl:stylesheet xmlns:xsl="http://..." ..>                2
  <xsl:template match="/">                                3
    <publications>                                        4
      <xsl:for-each select="//book">                      5
        <title>                                           6
          <xsl:value-of select="@title"/>                 7
        </title>                                          8
      </xsl:for-each>                                     9
    </publications>                                       10
  </xsl:template>                                         11
</xsl:stylesheet>                                         12
```

This template generates (once applied with appropriate instantiation data) for each `<book/>`-node `<title/>`-nodes with matching titles as text in its children nodes directly underneath `<publications/>`. Line 4 matches against the top-level element node — this means the whole document is considered. In line 6, all `<book/>`-nodes starting from the top of the document are determined. Now suppose the associated source document is:

```
<bibliography>                                            1
  <book author="Simon Thompson"                           2
    title="Haskell - The Craft ..."/>                     3
  <magazin title="Informatik-Spektrum..."/>               4
  <book author="Joshua Kerievsky"                         5
    title="Refactoring to Pa ..."/>                       6
  <url title="XSD specification 1.0"/>                    7
</bibliography>                                           8
```

In line 6 of the stylesheet, the nodes from lines 2 and 4 matches. Each of these nodes is successively considered the source document itself (see lines 7-9). While processing, first, the template element node `<publications/>` is copied, and second the instantiation of its child nodes is continued.

```
<xsl:for-each select="//book">                            1
  <title>                                                 2
    <xsl:value-of select="@title"/>                       3
  </title>                                                4
</xsl:for-each>                                           5
```

instantiates for the first element of the set:
C_1 = `[<book title="Haskell..." ... />,`
`<book title="Refact..." ... />]`
the node:

```
<publications>                                            1
  <title>Haskell...</title>                               2
  <xsl:for-each select="//book">                          3
    <title>                                               4
      <xsl:value-of select="@title"/>                     5
    </title>                                              6
  </xsl:for-each>                                         7
</publications>                                           8
```

In a second step the remaining set C_2 =[<book title="Refactoring..." ... />] is applied to the nested loop. The next node results:

```
<publications>
   <title>Haskell...</title>
   <title>Refactoring...</title>

   <xsl:for-each select="//book">
      <title>
         <xsl:value-of select="@title"/>
      </title>
   </xsl:for-each>
</publications>
```

with $C_3 = \emptyset$ as the remaining empty set. The loop condition does not hold anymore for the instantiation of loops in XSLT (cf. [19],[67]). So, the nested loop is skipped. Since no more following nodes exist, the instantiation terminates and returns this node:

```
<publications>
  <title>Haskell - The Craft ...</title>
  <title>Refactoring to Patterns</title>
</publications>
```

B. Validation

Validation checks if a given XML document is an instance of a template (see fig.1). If the answer is "'yes"', then the given document is "'valid"', otherwise not. Templates generate instances, which schemas may describe. That means template and schema de facto describe the same instance document. If we try to unify both processes, then both descriptions also need to be unified somehow. Otherwise, not the same template or corresponding schema may be expressed. One big problem in doing so is that both syntax and semantic differ in both cases. The template engine would need to be reconfigured, so documents are not expanded but schema-validated. In practice, this means validation is triggered rather than a template engine.

The *schema* describes the set of all valid instances of a template. Schemas often are tagged and, as such, are XML-dialects. Represents of such are *RelaxNG* [20] and *W3C's XSD* [28], [60], [7]. *DTD* represents a schema language for XML documents, but it is not XML. Schema languages in XML can be processed the same way templates are processed (cf. sect.III). XML-schemas consist of *literals*.

Literals are childless element and text nodes as well as composed element nodes. Composed element nodes have at least one child node and any number of commando tags in arbitrary order as children. *Commando tags* denote either a loop or a selection and may be decorated with *constraints*.

Different to instantiation, validation does not consider instantiation data. If instantiation data was complete, an instantiation could be performed. In that case, validation would be equal to checking the given instance document's equality with the instantiated document. If, however, instantiation was not complete, then conclusions would not always be correct. Validation could then be reduced to the problem of *document reconstruction* [31]. In practice, however, document reconstruction would induce too many burdensome restrictions, s.t. instantiation of a template would have to be invertible, for instance. That is why it is better not to make any assumption about the instantiation data instead.

The validation of instances may be considered as a *word problem* (cf. [16], [50]). In that case, the instance documents act as word and the template document as grammar, which generates all valid instance documents. In case the template contains cycles, the set of all valid instances becomes infinite, and instantiation becomes one possible derivation for grammar, which is the template document. If the document is interpreted as a term, the instance document represents a standard form if all slots are reduced until only element and text nodes remain.

C. XTL

Until now, schemas could not (satisfactory) be represented in the language of templates. A schema of some template language is too often too complicated to read or too complex. That is why XSLT templates require many helper functions and rewritings. Because of the properties an instance has to have, schemas can barely be automatically be inferred (cf. sect.VI). Despite that disadvantage, schema languages are still being used that do not unify smoothly with templates. As already mentioned, the differentiation between template and schema languages is why there is a considerable increase in maintenance and heterogeneous program systems. Another disadvantage may be additional efforts in learning new schema languages.

That is why the following points are required to unify template and schema documents:

- keep schemas short and simple
- each element in the template shall correspond with a similar element in the schema, and vice versa.

The XML-template language XTL was defined as part of the *SNOW*-project [59]. One of its goals was to investigate, if tractable at all, whether documents may be reused for template expansion and schema validation. XTL in version 1.0 currently counts seven intrinsic tags to be explained in more detail next. The comparison between template and schema nodes is taken out in sect.VI.

xtl:attribute

The insertion of new attributes can be expressed like this:

```
<xtl:attribute name="name"
    select="expression"/>
```

The tag does not have children and is under an element node [32]. Both 'name' and 'select' have to be specified as attributes. Attributes always have to be under an element

node. In case the same attribute name has already been defined, the new value *expression* replaces the attribute. The definition order of attributes is arbitrary and still defines the same element node. W.l.o.g. it is agreed that attributes are in *canonised* form. It implies attribute assignment pairs are ordered ascending by the attribute name in lexicographic order — attributes reference instantiation data by 'select'. In JXPath as template engine and PHP as a template language, the *expression* denotes a path expression.

```
<book id="1">
 <xtl:attribute name="author"
  select="//book[position()==1]/@author"/>
 <xtl:attribute name="id" select="999"/>
</book>
```

For this example, sect.II-A as instantiation data and JXPath as PHP, the node `<book author="Simon Thompson" id="999"/>` is instantiated. The first attribute definition inserts a new entry to the attributes list. Since XPath maps integers on themselves [11], the new attribute's value is `"999"`.

xtl:text

The childless tag for text inclusion is

`<xtl:text select="`*expression*`"/>`

Expression is passed to the placeholder plugin during instantiation, which will handle it further, such as XPath-expression. The resulting nodes list is converted by implicit coercion in XPath [11] into a string, which finally is concatenated. The concatenated text replaces the tag for XTL-text-inclusion.

The expansion of two neighbouring 'xtl:text' nodes as children is of interest, especially for validation. This interest comes from asking how to split a common string best when there are no markers that indicate boundaries. That is why the separation may become ambiguous. Hence, strategies are wanted, which allow recognising 'xtl:text' nodes uniquely. In contrast to this, 'xtl:attribute'-nodes do not have this problem.

`<xtl:text select="/"/>` instantiates the empty string for the example from sect.II-A, because the source document is traversed in pre-order and occurrences of text nodes are accumulated and concatenated.

xtl:include

The following childless tag can achieve an arbitrary element node enriched by instantiation data:

`<xtl:include select="`*expression*`"/>`

The PHP returns either one well-formed element node or none. If multiple nodes match with *expression*, PHP chooses only the first occurrence and drops all others [32].

The attribute *expression* equals `"//url"` returns for the example from sect.II-A the element node:

```
<url title="XSD specification 1.0"/>.
```

xtl:if

Conditions in XTL have this form:

`<xtl:if select="`*expression*`">...</xtl:if>`

If *expression* evaluates to "'true"', then the evaluation continues with its children '...'. Otherwise, the children nodes of 'xtl:if' are dropped, and evaluation proceeds with the following siblings.

An example determining the second book of a bibliography, if any, looks like this:

```
<xtl:if select="//book[position()==2]">
  <xtl:include select="//book"/>
</xtl:if>
```

For the element node `<bibliography/>` from sect.II-A it retrieves the following node:

```
<book author="Joshua Kerievsky"
  title="Refactoring to Patterns"/> .
```

xtl:for-each

The tag for cycles is

`<xtl:for-each select="`*expression*`">...</xtl:for-each>`

The evaluation of *expression* by the PHP returns a nodes list. This list is iterated successively, and each node is propagated as context for the instantiation of children nodes (cf. sect.II-C). The instantiation of children nodes with the first element from the evaluation of 'select' returns an instantiated children lists. The same goes for the ongoing instantiation. Those are linked together until no more context exists.

The use of a context does not restrict the reachability of axes because every node remains reachable. For example, nodes located in the upper section of a source document may by XPath [11] be addressed using 'ancestor'. Contexts are using also used in XSTL [19] for the sake of usability.

xtl:macro

Macros are defined in XTL as following:

`<xtl:macro name="`*ncname*`">...</xtl:macro>`

Macros are *symbols*, which bind arbitrary sequences of command, element and text nodes, except further macro definitions. The macro is defined by a fully qualified name *ncname* which must be unique among all macros within a template. In a template, all macro definitions must be contiguous and before a sequence of non-macro definitions right underneath the top element node [32].

xtl:call-macro

The macro call without children is defined as

```
<xtl:call-macro select="ncname"/>
```

A macro call is similar to a function call without parameters. The macro call retrieves a list of element nodes. During expansion and validation, the macro call is replaced by the right-hand side of element nodes from its definition. Recursive calls are permitted. Termination conditions need to be specified within '`select`'-expressions in XTL command tags.

Summary:

The following fragment of a XTL-schema demonstrates validation with macros and cycles.

```
1  <xtl:macro name="TDs">
2   <td>
3    <xtl:text select="@title"/>
4   </td>
5   <td>
6    <xtl:text select="@author"/>
7   </td>
8  </xtl:macro>
9
10 <table col="#FF0000">
11  <th>
12   <td>Title</td>
13   <td>Author</td>
14  </th>
15
16  <xtl:for-each select="//book">
17   <xtl:if select="position() mod 2=0">
18    <tr col="#333300">
19     <xtl:call-macro name="TDs"/>
20    </tr>
21   </xtl:if>
22   <xtl:if select="position() mod 2=1">
23    <tr>
24     <xtl:call-macro name="TDs"/>
25    </tr>
26   </xtl:if>
27  </xtl:for-each>
28  <tr>
29   <td>XSD specification 1.0</td>
30   <td/>
31  </tr>
32 </table>
```

A corresponding well-formed instance would be:

```
1  <table col="#FF0000">
2   <th>
3    <td>Title</td>
4    <td>Author</td>
5   </th>
6   <tr>
7    <td>Haskell - The Craft of Functio ...</td>
8    <td>Simon Thompson</td>
9   </tr>
10  <tr col="#333300">
11   <td>Refactoring to Patterns</td>
12   <td>Joshua Kerievsky</td>
13  </tr>
14  <tr>
15   <td>XSD specification 1.0</td>
16   <td/>
17  </tr>
18 </table>
```

First, validation stores the macro '`TDs`' defined on lines 2-7 and continues from line 10. This line matches with line 1 of the instance. The child node at lines 11-14 entirely matches with the child node of lines 2-5 from the instance. At lines 16-27, there is a non-deterministic decision to be made on whether child nodes match for a cycle. It is impossible to determine how much further a cycle needs to be unrolled to match the schema in the instance document at line 6 without checking the following nodes. If the cycle '`xtl:for-each`' is left too early or too late, then the `<tr/>`-node from lines 28-30 may not exactly match with the expected number of nodes from the instance. According to the instance document, '`xtl:for-each`' may have no, one, two or three iterations. It is necessary to continue on fails with alternatives, if there are any, to guarantee a correct validation Only after all alternatives fail, validation fails.

In the previous example, the correct number of iterations, which is two, is guessed, s.t. both `<tr/>`-nodes from lines 6-9 and 10-13 from the instance match consecutively with lines 7-16 from the schema. '`select`'-expressions from the conditions are ignored here. Consequently, shuffling `<tr/>`-nodes in the instance document, but also a sequence of colored `<tr/>`-nodes lead to a true validation.

The instantiation of the cycle is interpreted as unrolling all books from the instantiation data. Instantiation continues with the following tags. In analogy to that, validation tests if the last node of the instance from lines 14-17 matches `<tr/>`-nodes from the schema at lines 28-31. As this is the case and no other nodes follow, the validation quits successfully.

D. Theoretic Foundations

This section introduces theoretic foundations. First, the tree-structured data model "'hedge"' is defined, then regular tree grammars and languages. Later a short overview is given on regular automata. Examples illustrate definitions.

1) Trees: The theories introduced later in this section may be applied to tree-structured objects, like XML schemas, XML instances and instantiation data. XML documents can be represented as trees since nodes of an XML document are in a hierarchy, and there are no cycles included all through ascending edges leading from leaves to the root element node. Keys as used to describe relations of a schema are not of interest regarding a schema's syntax.

XML documents are multi-way trees with child-rich elements as nodes and childless elements, and text nodes as leaves. Attributes of element nodes may be transformed into element nodes with new child nodes that represent such attributes. For example, the node `<a id="1"/>` can be transformed into `<a><id>1</id><sep/></a>`, where

<sep/> separates attribute nodes from original child nodes not representing former attributes.

Before trees and their properties are introduced, their practical meaning is recapitulated.

XML documents can either be interpreted as unstructured, namely as text, or structured. By interpreting XML as unstructured text, important information vanishes, for instance, newlines or ordering. A replacement of element nodes in trees by symbols returns trees. Replacements may cause shorter nodes sequences. That is the reason why existing string-grammars are going to be extended and reused (see sect.II-D2, cf. [50], [12]).

In structured XML-interpretations tags emphasise text regions. Tags denote meta-information and are not part of the document text. Hence, XML documents are predestined for structural interpretation, for both instantiation with command tags, and validation with a common element and text nodes. For example, the unstructured interpretation of:

<a>helloworld<c/>

does not allow a simple processing neither by a user nor by a program. The interpretation of its structure makes access to the documents' content easy.

Definition 2.1. *A hedge (after Murata [50]) is defined over a finite symbol set Σ and a finite variable set X as following:*

ε	.. *the empty hedge*
x	.. *variable with* 'x'$\in X$
a<u>	.. *element node* 'a'$\in \Sigma$ *with hedge* 'u'
u v	.. *concatenation of hedges* 'u' *and* 'v'

.□

The two XML-element nodes <a/> and x are representable as hedge a<ε>b<b<ε>x> with the symbol set Σ=$\{a, b\}$ and variable set X=$\{x\}$. Σ does not oblige any restriction. Element names may have any prefix and suffix. Hence, every XML document is representable as a hedge, even XTL-templates and XSLT stylesheets.

Based on this model, each navigation operator over trees can be defined (see [49], [31], [11]). For instance, the function subtree *[49] distinctively determines a predecessor node for a given number-encoded path.*

2) Regular Tree Grammars: As previously mentioned, string grammars are not sufficient to describe trees. Hedges do not only grow in width by concatenation, but they also grow into depth by insertion of child nodes. Regular tree grammars are suggested as one way to resolve this issue.

Definition 2.2. *A regular hedge-grammar (RHG, after Murata [50]) is a grammar $G = (\Sigma, X, N, P, n_f)$ with*

Σ	.. *finite symbol set*
X	.. *finite variable set*
N	.. *finite non-terminal set*
P	.. *production rules*
n_f	.. *final state set.*

A production rule from P has either the form $n \to x$, where $n \in N$, $x \in X$, or $n \to$ a<r>, where a $\in \Sigma$ and r is a regular expression over $N \cup X$ herewith. n_f denotes a regular expression over N, which is accepted by the grammar. □

Definition 2.3. *Regular expression over hedges.*

Let r, r_1, r_2 be regular expressions over a finite set of non-terminals. Then the following expressions are also regular:

$r_1 \cdot r_2$	.. *concatenation*
$r_1 \mid r_2$	.. *alternative*
(r)	.. *parantheses*
r^*	.. *repetition*

□

Regular expressions are better for XML-schemas than relations or rigid associations, as demonstrated by [16]. Relations do have a fixed amount and order of arguments. Contrary to this, regular expressions allow short but flexible expressions.

The form of its productions also demonstrates the regularity of RHS. The set of valid instances for a XTL-schema:

```
<xtl:if select="//checked">
 <a>
  <xtl:for-each select="//person">
   <x/>
  </xtl:for-each>
 </a>
</xtl:if>
```

recognises the regular tree language

$$L(G) = \{\varepsilon, a < \varepsilon >, a < x >, a < xx >, a < xxx >, ...\}$$

(cf. sect.II-D3). A corresponding grammar G is $G = \{\Sigma, X, N, P, n_1?\}$ with Σ=$\{a\}$, X=$\{x\}$, N=$\{n_1, n_2\}$ and P: $n_1 \to a < n_2^* >$, $n_2 \to x$.

RHG differs from string-grammars in variables, which can be considered terminals and a final state set, describing the accepted language. Productions are similar to regular productions, whereas the right-hand side of each rule may contain further non-terminals of the form $e_0 \cdot e_1 \cdot ... \cdot e_n$, where e_n is a non-terminal and all other e_j for $\in [0..(n-1)]$ denote terminals. Terminals stand solely or to the left of a non-terminal.

Derivations for regular tree grammars work similarly to string-grammars. Non-terminal symbols are derived left-to-right until the derived expression does no more contain non-terminals.

The derivation of a regular tree grammar corresponds to the instantiation of an XTL-template.

The derivation of regular grammars may be non-deterministic due to multiple rules for selection. If there exist at least two derivations of a non-terminal, then an implemented automaton may not decide in general without a stack (cf. [16], [63], [9]).

Regarding grammars, the question concerning expressibility emerges, for instance, if a tree grammar is context-free or context-sensitive. Context-free grammar does not leave open questions from a practical standpoint (see sect.III). However, context-sensitive grammars are not that easy. That is why,

often, contextual information is transmitted by a different mechanism. Turing-mighty tree grammars are not further considered here.

Murata, Lee, and Mani [48] categorise regular tree grammar's expressibility as following:

$$local \subseteq single-type \subseteq ranked-competing \subseteq regular$$

The ordering is due to the level of non-determinism of the production rules (ambiguity). Local tree grammars are the weakest. Regular tree grammars are the most powerful. More powerful grammars entirely contain weaker grammars. Moreover, more powerful grammars always contain non-empty cases which are not covered by the weaker grammars [48]. Four tree grammars may be categorised as follows:

Local: *A terminal may not occur in more than one rule.*

S.-t.: *Non-terminals of children nodes do not compete with each other, which means $\pi(e_i) \cap \pi(e_j)$ is empty for each two distinct nodes e_i and e_j. $\pi(e_j)$ determines the set of possible beginnings for some node e_j.*

R.-c.: *A hedge r is uniquely decomposable. It means $\forall U, V, W \in N : r \nvdash_* UAV$ and $r \nvdash_* UBW$ for competing non-terminals A, B in r.*

Reg.: *All regular grammars which are not ranked-competing.*

The following two examples explain the membership of a specific tree grammar:

Ex 1 *Let the grammar G_1 have the following productions:*

$Doc \rightarrow doc(Para1, Para2^*)$
$Para1 \rightarrow para(Pcdata)$
$Para2 \rightarrow para(Pcdata)$
$Pcdata \rightarrow pcdata\ \varepsilon$

$Para1$ and $Para2$ compete with each other in the first production. Hence, for a ranked-competing tree grammar no U, V, W may exist, s.t. $r \vdash_* U\ Para1\ V$ and $r \vdash_* U\ Para2\ W$, where $r = Para1\ Para2^*$. Since U must be different in the first derivation from the second, we just found a contradiction. Because no other decompositions exist, G_1 is ranked-competing and therefore regular too.

Ex 2 *Let the grammar G_2 have this productions:*
$Doc \rightarrow doc(Para1^*, Para2^*, Pcdata)$
$Para1 \rightarrow para(Pcdata)$
$Para2 \rightarrow para(Pcdata)$
$Pcdata \rightarrow pcdata\ \varepsilon$

$Para1$ competes with $Para2$ in

$$r = Para1^*\ Para2^*\ Pcdata$$

. Hence, no U, V, W exist, s.t. $r \vdash_* U\ Para1\ V$ and $r \vdash_* U\ Para2\ W$. But, there exists the valid decomposition $U = \varepsilon$, $V = V'\ Pcdata$, $W = Pcdata$. That is why this grammar is not ranked-competing.

3) Regular Tree Languages: Regular Tree Languages are formal languages generated by RHG, hedge-regular expressions and deterministic and non-deterministic hedge-automata (see [49]).

The transformation between the models mentioned above is very similar to those in string-based formal languages. Hedge and tree models and grammar and expressions are from a computability perspective equivalent — this is shown in [49]. Trees are specialised hedges, and a hedge is a tree with an empty root node.

It is worth mentioning that element and text nodes can be treated nearly the same (cf. sect.II-D1). As shown earlier, attributes may be simulated by element nodes. Examples to each of the mentioned models in this section may also be found in [49], [50], [16], [4].

4) Finite Tree Automata: Regular automata are being addressed in [50], [12]. Next, only such automata are characterised, which correspond to the expressibility of regular tree grammars. The taxonomy proposed in [48] is used to achieve this goal, particularly the determinism and evaluation order is of utmost interest and is summarised in tab.I. Bottom-up automata can recognise ranked-competing grammars.

	deterministic	non-deterministic
top-down	local, single-type	regular
bottom-up	ranked-competing	regular

TABLE I
TAXONOMY OF TREE AUTOMATA

According to [48], grammars whose languages are recognised by non-deterministic top-down and bottom-up automata are reducible to the same. Non-deterministic grammars and recognition algorithms are often significantly shorter in their description than equivalent deterministic grammars. However, those algorithms may be more complex and involve extensive backtracking. In contrast to top-down automata, Bottom-up automata are considerably more complex and, therefore, more challenging to maintain. Especially, error messaging may become more complicated by far since, in general, all alternatives need to be checked before deciding if validation fails.

Moreover, the "'real"' reason would have to be tracked somehow among alternating backtraces.

Derivatives [9] is one approach, which maps regular expressions over hedges onto finite regular tree automata. During a document validation, a given schema and regular expression are reduced towards an instance until both sides cannot be reduced any further. If both expressions are empty, then validation would succeed. Otherwise, validation fails. After each derivation step, a new state is introduced. Since the

incoming schema is finite and symbols are not allowed (see sect.II-C), the algorithm terminates.

Partial-Derivatives [4] is an improved approach that calculates derivatives only when needed (see app.X). Once calculated, solutions are not determined a second time. Similar parsing approaches increase performance by 70% in XML-documents [43] and 80% in instantiating those [51].

The additional cost to be paid on XML-parsers is only causing approximately 10% of overhead. This algorithm has a best-case complexity of $\theta(n) = n$ and $O(n) = n^2$ for the worst case, where n is the given regular expression length. For the reasons mentioned, this approach is of interest for practical implementations. It can be stated that tree automata describe functions over XML documents, particularly the template expansion and schema validation.

III. Analysis

This section investigates instantiation and validation. Both functions are considered and requirements formulated for an unification on document level.

A. Current Situation

This paragraph XTL is analysed w.r.t. language features. Then instantiation and validation are investigated closer, e.g. properties of XML-template and schema languages.

1) XTL: ***First Considerations***

Represents of template languages are JSP, ASP, XSLT and XTL. Prolog may also be considered for instantiation of XML-document [31].

The generated target language may distinguish template languages. If both template and target language unite, as may be the case with XML, it is obvious that unification may be easier.

Its intention may also distinguish template languages. For instance, template languages with variables and functions are more appropriate for programming than for document processing. In any case, it is worth, to separate the program from the document, especially when it comes to validation (cf. sect.III-A2).

Parr [52] proposes template languages to have the following minimal asset of template commands, which should also count for document processing:

1. *Attribute references,*
2. *Conditions,*
3. *Recursive Template Calls,*
4. *Conditional Template Inclusion.*

XTL, which is also a template language, already has these features (cf. sect.II-C). '`xtl:attribute`' express attributes, conditions by '`xtl:if`', cycles by '`xtl:for-each`', text inclusions by '`xtl:text`' and element inclusions by '`xtl:include`'. Applications can be expressed, but only without parameters (done by '`xtl:call-macro`').

XTL instantiates free of side effects. So, queries to instantiation data do not alter the source nor the template. This way, referential transparency is guaranteed.

If PHPs allow access to documents, then XTL indeed is also favourable for documents. Helper functions should be banned from XTL in general and moved to external sources, which may be referenced by '`select`'.

Separation of Concerns

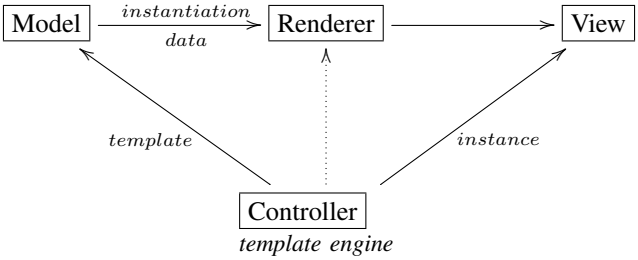


Fig. 2. Model-View-Controller for Instantiation

Czarnecki and Eisenecker [14] insist a transformation is taken out by referencing only a minimal set of operators. That also goes for instantiations because these are transformations too. Both suggest restricting ourselves to loops and selections instead. In XTL, this is done by the command tags '`xtl:for-each`', '`xtl:text`', '`xtl:include`', '`xtl:attribute`', and the attribute '`select`'. However, functions shall better be removed from instantiation and be passed to a module responsible for that particular task. For the same reason, calculations shall also be removed from templates. It also affects arithmetic expressions, e.g. in XPath, which are not needed for navigating instantiation data.

Furthermore, Parr [52] insists template languages separate between concerns of a given instantiation problem. Fig.2 shows a concern separation for XTL using the Model-View-Controller meta-pattern after Reenskaug. The resulting instance document shall be independent of the instantiation, s.t. all calculations and existing constraints fit within the chosen model. Output in View often requires formatting. Therefore, the Controller triggers the Renderer, who passes information further from Model to View. By doing so, the strict separation remains.

The separation of command language from the template language by '`select`' and common tags lead to an increasing similarity between instance and template. It is the foundation for the unification of instantiation and validation (see sect.VI-B).

The separation between template and command languages causes the command language has to provide conventional interfaces for text and attribute access, inclusions, conditions and loops. Since XTL does not allow direct access to instantiation data initially, it is not Turing-mighty. The optional attribute '`realm`' was introduced. In XTL command tags, it may be inserted by a '`select`'-attribute in order to allow XTL to access different instantiation data.

Type Safety

XTL guarantees a well-formed document during instantiation of a template if it is sound w.r.t. its specification [32].

Each XTL-template is well-formed XML. The only restriction is the top root node may not be a command-tags according to its specification. It must be an element node. All nodes located underneath are well-formed because those nodes are composed of text and element nodes only. In some cases ‘`select`’*-queries are not sound. The affected XTL-command are left empty by the XTL template engine.*

PHP functions assert type safety. Hence, instances are always type-safe here. Static types of PHP functions make instantiation more predictable before running. It means command languages become exchangeable, and so the source language becomes more flexible.

In XTL, term-evaluator [33] and instantiation data evaluator have been introduced to evaluate ‘`select`’*-expressions. The instantiation data evaluator checks the types of a solution with the inferred type or an optional type. Instantiation data has a polymorph type and so have query results. In order to process arbitrary instantiation data, polymorphic data needs to be transformed into non-interpretable text. That is done by the Renderer, which knows about desired output formatting.*

The following table shows the types of XTL-commands, which need to be known to the Renderer before building the instance document:

```
xtl:attribute  ::  String → [a] → String
  xtl:include  ::  String → [a] → XML
     xtl:text  ::  String → [a] → String
       xtl:if  ::  String → [a] → Bool
 xtl:for-each  ::  String → [a] → [a]
```

The first type denotes an `select`*-expression. The second type denotes the instantiation type, which is polymorph. The typing of PHP-functions is explained in more detail:*

```
<book>
  <xtl:attribute          SELECT title
    name="title"     ⟵    FROM books
    select="..."/>       WHERE id='1'
</book>
```

It is assumed, a PHP function determines for a given SQL-query a relation with exactly one result. This result is converted by the Renderer in a representable string and placed into an instance document. The example generates a `<book/>`*-node with attribute* `title="Haskell..."` *related to the example from sect.I. Though, the resulting nodes are always well-typed and well-formed for any instantiation data and PHP functions. So, every XML document created is type-safe.*

Variability

It means the exchange of the command language. PHP has standard interfaces and concrete implementations. By doing so, the internal organisation can be hidden from the user [6]. The processing of heterogeneous data is managed by previously agreed interfaces only.

The template engine restricts access to instantiation data. The template may not grant access. Although this rigid restriction may not always be desired, as demonstrated by the following fragment:

```
<book>
  <xtl:include select="document('a.xml')
   //title"/>
</book>
```

The difficulty is the template addresses external documents during instantiation. The function ‘`document`’ *is evaluated with the path expression following, even if the function is not XPath [11]. The source is passed immediately to the instantiator. Alternatively, all referenced spots are moved to a separate document to resolve this problem — this is tractable only if there are multiple sources. In the template, those entries are referenced. Moreover, the extraction of all needed sources by additional templates is in preparation.*

Style

Besides the formal grammar type, schema languages can also be characterised [46] by the corresponding grammar style.

A language has a grammar style if the associated schema language is described shorter by a grammar than by a corresponding regular expression [18] (see following sections). Otherwise, the language is in a pattern-based style.

A schema language allows many different tags at different locations. Schemas that only have a few restricting symbols only are more often in pattern-based style than in grammar-style.

Macro definitions can easily be described in XTL in grammar style. Right-hand sides are equal to a not necessarily right-ranked hedge (cf. sect.IV). All remaining XTL-tags may be described in pattern-based style as well as in grammar style.

In contrast, we have schemas from rigid associations/relations whose elements need to be placed separately. Relations do not insist on strict ordering. However, related entities must obey certain conventions. For example, relations need to be defined a priori, which determine uni-directional element and bi-directional attribute relations. It also causes a whole graph is represented.

Regularity

Since RHG may be described by XML-schema languages [16], [50], a corresponding representation exists consisting of regular hedge-expressions. If ‘`select`’*-expressions, which denote the number of repetitions, in* ‘`xtl:for-each`’*-loops are kept arbitrary, then context-sensitive and context-regular expressions become regular expressions for the price of further abstraction. Arbitrary repetitions become Kleene's star operators (cf. sect.III-A3). Conditions in XTL are represented either by ε, hedges, or text and element nodes as literals.*

Macro definitions and macro calls require special treatment because right-hand sides of macro definitions may contain an arbitrary number of further macro calls anywhere within the hedge. Both extend the expressibility of regular tree grammars. Therefore, macros need to be investigated when it comes to judgements about expressibility. After all, and for the sake of simplicity, it still makes sense to regard XTL as

```
<xtl:macro name="M">
 <a/>
</xtl:macro>
<a/>
```
(a)

```
<xtl:macro name="A"/>
  <a/>
  <xtl:call-macro name="C"/>
</xtl:macro>

<xtl:macro name="B"/>
  <a/>
  <xtl:call-macro name="D"/>
</xtl:macro>

hello
<xtl:call-macro name="A"/>
world
<xtl:call-macro name="B"/>
!
```
(b)

```
...

<xtl:macro name="A">
 <xtl:for-each select="//A/book">
   <a/>
 </xtl:for-each>
 <xtl:call-macro name="C"/>
</xtl:macro>

<xtl:macro name="B">
 <xtl:for-each select="//B/book">
   <a/>
 </xtl:for-each>
 <xtl:call-macro name="D"/>
</xtl:macro>

hello
<xtl:call-macro name="A"/>
<xtl:call-macro name="B"/>
world!
```
(c)

TABLE II
EXPRESSIBILITY OF REGULAR SCHEMAS IN XTL

a regular schema language. A more detailed investigation of the grammar class follows.

Example (a) from tab.II shows that XTL is not only local because `<a/>` *occurs in the body of the macro and in the hedge. It means the corresponding grammar has two productions with the exact right sides. Example (b) shows XTL is not only of single-type. Macros '*`A`*' and '*`B`*' are competing – both contain '*`<a/>`*' as a starting symbol. Hence,* $\pi(A) \cap \pi(B)$ *is not empty. Example (c) shows XTL is not only ranked-competing because decompositions cannot always be found due to uneven macros '*`C`*' and '*`D`*'. Hence, XTL is as expressible as the class of languages generated by regular tree grammars.*

Only regular tree languages are enclosed under union, intersection and complement (cf. [16]) as string-grammars are. That is particular of interest when extending XTL by command-tags composed of existing tags.

Those tags oppose non-monotone operators [31] because those may change instantiation data fragments and those do not necessarily merge existing instantiation data. Those operators are advantageous and compact when an instance shall have many details and when the difference between instance and source documents is relatively small. Rather than requesting much information to build up the document from scratch by filling numerous slots, it may be more efficient to copy the document instead nearly. Although non-monotone operators can reduce a template significantly, violated closure properties may cause disturbance in the separation of concerns between instance and instantiation data because validation can not make assumptions about the instantiation data. That is why non-monotone operators must be restricted in regular languages a priori not to hinder the unification of instantiation and validation.

Context-free Tree Languages

Schemas can be defined in two ways (see tab.III) to recognise the context-free tree language $L(a^n b^n)$.

Variant 1:
```
<xtl:macro name="S">
 <xtl:if select="...">
  <a/>
   <xtl:call-macro name="S"/>
  <b/>
 </xtl:if>
</xtl:macro>

<xtl:call-macro name="S"/>
```

Variant 2:
```
<xtl:for-each select="//book">
  <a/>
</xtl:for-each>
...
<xtl:for-each select="//book">
  <b/>
</xtl:for-each>
```

TABLE III
CONTEXT-FREE SCHEMAS

Variant 2 is universal since $L(a^n c b^n)$ *could be generated. That cannot be done with variant 1. Furthermore,* $L(a^n u b^n v c^n)$ *and* $L(x a^n v b^n v c^n w d^n y)$ *can only be recognised by variant 2. So, the difficulty is to express exactly instances that shall be recognised during validation. For instance, if '*`select`*'-expressions describe* $L(a^n c b^n)$*, then this language would be context-free. In contrast to regular language recognition, the recognition of programming languages, especially those defined by a LL(k) or LR(k)-grammar [30], is much more complex. Therefore, the recognition of context-free and context-sensitive tree languages, in general, might be much more complex and is doable but only with much more efforts to be spent. XTL is regular if only expressions of the command languages are not considered exactly during validation. Variant 1 describes a part of context-free*

schemas. However, if $L(a^n c b^n)$ is to be recognised, then variant 2 would need to be chosen. Nevertheless, context-free languages may not be validated exactly. This means `<a/><a/><a/>x<b/><b/>` *is enclosed by $L(a^n x b^n)$ and is validated by variant 2, although the input is not context-free. Hence, context-free schemas are not going to be considered further.*

Termination

If in the body of macro definitions appear macro calls, recursion may appear. Fixpoints may appear in the template and schema. Fixpoints may be formulated with the command language and remain unreachable due to left-recursions – so instantiation and validation do not terminate. However, the recognition of left-recursion is in general not decidable for XTL-documents due to the Halting problem.

In contrast, it may effectively be decided whether, for instance:

```
<xtl:macro name="M">
  <xtl:call-macro name="M"/>
</xtl:macro>
```

contains a non-terminating cycle. However, this does not work for arbitrary XTL-document before executing the program.

Functions

XTL has a small vocabulary. Because XTL does not have parameters, formal functions cannot be defined. In XPath, it is not possible to define functions with an arbitrary arity. It is also not possible to call such a function from other tags. A locally defined function may not violate the template engine's referential transparency because of the strict separation between template and instantiation data. So, many functions, mainly μ-recursive and tail-recursive functions, cannot be expressed within XTL.

Tail-recursive functions can be simulated in XTL but only under additional restrictions. For example, counters may be expressed by special '`select`*'-expressions. The function '*`position()`*' allows accessing the actual counter in XPath. The amount of loop iterations is limited in '*`xtl:for-each`*' by a constant value evaluated by '*`select`*'. Cycles simulate tail-recursive functions in XTL but without an argument list.*

Due to the lack of defining and composing functions, XTL is not primitive-recursive — even when while-loops are replaced by recursion with a preceding '`xtl:if`*' or when tail-recursive functions without arguments are mimicked. Due to the separation of concerns, document processing does not require sophisticated arithmetic nor logical functions.*

2) Instantiation: Instantiators can be interpreted as term rewriting systems. The λ-calculus provides a mechanism for doing so (see [5]). Slots can be represented as variables and nodes as terms. Then instantiation equals a derivation.

However, terms need to be adequately modelled. It may be needed to assign each element node with a certain arity its semantics. An evaluated node may denote a node with a qualifying functor, which encodes the number of children. Otherwise, a concatenation of nodes may not be injective.

Furthermore, empty nodes, attributes and hedges need to be mapped, which may require additional handling.

Values of '`select`*'-expressions are bound to variables. Within terms of λ-abstractions, this is done by applying expressions to slot variables. The internal evaluation by PHP functions remains invisible.*

Termination is equal to reaching a normal-form. Reduction is strictly monotone. Once evaluated, nodes are not reverted. The number of evaluation steps is polynomially bound, except macros. Macros may lead to self-application because macro bodies may reify variables. In that case, no normal-form could be found, so the template engine would not terminate. The evaluation sequence of a hedge does not matter since all hedge nodes have to be evaluated and independent. XTL conditions terminate when evaluated outside-in. However, they do not terminate in general in the opposite direction.

As already mentioned, slots return strings and nodes. That is why types over nodes and slots make sense.

Atomic element nodes represent their type themselves and do not require additional conventions. Atomic nodes may directly be passed to instance document and do not cause any side effects. Hence, a formal description of the instantiation benefits from denotational semantics.

The simple untyped λ-calculus is not sufficient here. The typed λ-calculus is needed for a formal description of instantiation. It is because types and constructors are helpful to the description of element and text nodes. Constructors denote parametrised types whose type variable is typed again. Types are composed of other types. On the other side, the simple λ-calculus does not provide a compact notation for function calls. Fixpoint combinators are the only possibility to mimic recursion.

In contrast to that, the denotational semantics allows us to express constructors, types and formal function in a meta-language. So, it becomes possible to express pattern-matching compactly and to use combinators too. The construction of nodes should be done according to minimisation criteria [31].

3) Validation: In analogy to parsers, validators check for a given programming language if an incoming XML document (program) is a valid instance of a given schema (grammar). It is more appropriate to refer to matchers [57] when validation is meant. Compilers have an invariant set of rules but are applied to ever-changing incoming programs. When considering validators not only do the input data (instance) change and the set of rules (schema). Moreover, there is no translation going on, but a boolean value is calculated.

An XTL-validator checks an instance document node-wise against a regular schema. In [57], the classification of schema-matchings is proposed. The XTL-validator is hence a class on its own and is schema-centric. The validation operates on hierarchic XML-nodes and therefore is structure-centric. Both '`xtl:for-each`*' and '*`xtl:if`*' are meta-operators that influence the processing of an instance document. On a lingual level, they are constraints, which have nothing really in common with '*`select`*'-expressions. That is why the validator belongs to the class of graph-matchers.*

The description of validation can either be described as textual or graphical. The relation between matching documents can be expressed by $\cong (s_1, s_2)$, where s_1 denotes a schema node and s_2 an instance node. Schema nodes matching with some instance node s_2 generate a set S_1, which in general is not singular. If S_1 is indeed not singular, then validation becomes valid and non-deterministic. S_1 cannot be empty since it at least contains s_2.

The validation problem can be interpreted as typing problem [5]. In [69], typing is proposed as a static validation approach using Haskell's built-in type system. If a document validates, then a type can be inferred. Otherwise, Haskell shows up a typing mismatch. The validation takes place without an actual validation algorithm by doing so. This approach only works with dedicated constructors used for constructing in Haskell a whole XML document.

The formal notation can be based on the λ-calculus, but the formalisation for validation is problematic since there seems to be no good representation adequacy.

Context-free languages are not recognised, except the feature discussed in sect.III-A1. It means hedges are invalid as soon as they appear twice. The number of opening and closing brackets is only of minor interest – this is different from programming languages. A context-free schema may still be recognised by moving a sequence c to a prefix from the language $L(a^n c b^n)$ or moving c to a suffix of $a^n b^n$. In node c n may not occur. Otherwise, the given schema is not context-free.

To improve the runtime behaviour towards non-deterministic decisions determining a tree automaton requires a regular schema whose recognition has a complexity of $O(2^n)$. Herewith, n is the cardinality of the set of states (see [9],[4], cf. sect.II-D4). For validation to be used only once, these may mean too high costs. However, if many documents are going to be validated against the same schema, then the situation becomes different, as it may be the case with database triggers.

As seen in sect.II, tree automata fit only for schemas that do not change over time.

Clark [17] proposes a top-down non-deterministic algorithm for RelaxNG-schemas. Non-deterministic matchings are resolved by so-called interleavings [17]. He proposes rules for element nodes for all possible occurring cases. Inclusions denote variable symbols which arbitrary nodes may replace. It causes nodes to appear valid, although they do not occur at corresponding positions in the instance document. So, instead of an ε-node, its successor may be taken for validation, and if validation fails, then everything from there backwards needs to be analysed manually, which is quite laborious. It may be more efficient to track all possible nodes during validation and decide when to include the next time. Unfortunately, even small documents generate such a vast search space, so it becomes not doable due to an exponential rise in complexity. The extensive search [17] should if used at all, massively reduce invalid nodes. There is no optimal solution for this problem. However, there exist a few heuristics which may overcome the practical problem for previous domain ranges, such as the strategy "'try all valid states until a contradiction occurs"'.

Antimirov [4] proposes an algorithm turning a regular expression into a non-deterministic finite automaton (NFA). Hence, XML schemas can be considered as regular expressions. Antimirov's approach matches regular tree expressions also. Non-deterministic finite automata are dual to deterministic finite tree automata (cf. [49],[48]). In contrast to [9], needed derivatives only are calculated, and those are calculated only once. In procedural and object-oriented programming languages, the determination can be achieved by merging non-determined states or balancing non-determinism, e.g. by backtracking.

An alternative to validation (with XTL) is the transformation (of XTL) into another already existing schema language, like RelaxNG. Problems that need to be addressed could, but do not necessarily need to be: restriction of expressibility (of XTL) or coverage of the schema-language to be replaced. A schema transformation would also require additional well-defined schema languages, which is not part of this work.

4) Properties: Instantiation, as introduced in sect.II, is a mapping whose co-domain XML is entirely covered. Instantiation can be interpreted as endomorphism because both domain and co-domain denote the same set, namely XML. The well-formedness of XTL itself guarantees this. Therefore, instantiation is an enclosed operation (cf. [54]).

If instantiation is indeed considered an operation, then associativity does not hold. Commutativity also does not hold because instantiation of a slot-containing template document has XML as a result. The result syntactically does not match in general with the origin template. Instantiation of an XML document without slots is idempotent for any instantiation data.

For validation, particularly a derivation of a regular expression, homomorphism holds. It follows from this equation from [4], which also holds for schemas:

$$val(x \cdot y) = val(x) \odot val(y)$$

The operator '$\cdot$' concatenates two regular expressions and '$\odot$' logically ANDs two interleavings. So, some regular expression $val(x \cdot y)$ is congruent to $val(y)$ modulo $val(x)$ (see sect.IV-A4). Therefore, the order does not matter, whether first regular subexpressions x and y are evaluated and concatenated second, or whether first those expressions are concatenated and second x and y are evaluated. It might be helpful when at least one subexpression may be dropped, for instance, for seeking optimal solutions.

5) Arrows and Filters: An arrow is a generalised monad (see [38], [39], [58]). It encapsulates functions as parameter. An arrow in Haskell is an instance of the class `arr` with two

functions:

```
arr::(a→b)→arr a b
>>>::arr a b→arr b c→arr a c
```

Like variables in programming languages, functions may also be made available in dedicated environment scopes and namespaces. Particularly for lazy parsing and serialisation, certain sets may be evaluated partially, allowing higher usability.

Instantiation may use macro definitions instead of arrows. However, during validation, macro definitions shall be avoided because apart from a macro environment, further semantic fields may be required, e.g. a list of all valid element nodes — which was decided not to research further for the sake of previous outcomes in this work.

Filters denote functions having a polymorph type `a→[a]`*, where* `a` *denotes a type variable. Filters are functions with an input vector and an arbitrary output vector. They can be classified according to their behaviour, for instance, by common combinators (cf. [69]) and functions not in typical combinator representation. The class mentioned second are functions whose head is specified using pattern matching. If possible, implementations should make use of pattern matching — the same as semantics make. By doing so, redundant iterations of trees are avoided, and structural definitions can be reused. Both effects increase readability. The elimination of multiple iterations cost high efforts (cf. [64]), causing an increase in complexity. The gap between denotational semantics on the one side and implementations on the other side diminishes by specifying pattern matching.*

Unfortunately, both XML-parsing and serialisation, violate referential transparency, but this must be since files can neither be read nor written to without side effects. Luckily, these are the only places where this is required, and there is no other place having this effect. As an alternative to arrows, multi-paradigmal programming may be an option (cf. [24],[31]). By violating the absence of side-effects, read and write operations increase the flexibility of a function in general. Input and output operations are no longer restricted to a certain location in a function but now can be located and used anywhere. Multi-paradigmal programming shall be used, s.t. input and output operations are implemented by machine-dependent instructions, and where an abstract programming language implements instantiation and validation. Here, a violation of encapsulation would increase usability. Named functions shall be strictly typed and be implemented as super-combinators.

B. Requirements

This work aims to attempt to unify both views, instantiation, and validation (see sect.II). In consequence, functionality increases.

For demonstration purposes, denotational semantics would be required for both instantiation and validation. Algorithms based on it, implementation and an object-oriented design with test cases are prepared.

1) Limitations: No assumptions on a concrete command language are agreed upon that. So, no information on the syntax nor internal states, nor properties are known to the instantiator. No assumptions on the structure of instantiation data are made. Only PHP functions with previously agreed interfaces are to be considered (cf. sect.IV-B2). Communication is established exclusively by these interfaces.

Validation does not interpret '`select`*'-expressions. Bypass- and '*`realm`*'-attributes are not treated on instantiation. The support of '*`bypass`*'-attributes is optional during instantiation because this requires an the interference of several expansions within one template. The effect of '*`bypass`*' can be simulated by running several templates sequentially. '*`bypass`*'-attributes are then redirected as command tags into the instance document or as another '*`bypass`*'-attribute with a smaller value on the total amount of phases to be run [32].*

No precautions are made w.r.t. detect of non-terminating loops since this problem is in general undecidable (cf. sect.II-C).

Encountered problems of XTL in comparison to other schema languages are to be examined, and improvements shall be shown. The goal herewith is a compatible syntax extension (cf. sect.II-D3).

Another helpful tool for XTL is a semi-automated schema-generator, which generates a schema from an instance document (so-called "'validation by instance"'-approach) [26]. The implementation of a schema-generator for practical use would go far beyond the goal of this work. Therefore, it is not considered here further (cf. [26], [41]). Here are some reasons why:

- ***Parameter:*** *A schema appears useful to the user whenever it is fine-grained and recognises many regular substitutions in the instance.*
  However, since it is not obvious if a hedge may be replaced by a sequence of nodes or by a cycle with conditions, some criterion must be defined regarding granularity. So, complex substitutions could reduce an instance by a line, for instance, but the obtained instance would be tough to check by the user.
- ***Ordering:*** *Is the ordering within a children list fixed, or may it permute?*
  The ordering of a hedge is either explicit or selective. The ordering of nodes in a hedge, also referring to the following hedges, can be assigned by any attribute. Childless nodes do not necessarily need to be defined in a schema to be childless. Child-rich nodes may confirm in the following nodes the exception. Such differences rest exclusively on the user and may not be considered automatically.

- ***Quantity*** *Multipliers for specifying elements that occur several times are hardly available. Neither makes it sense to specify multiplicity on each occurring hedge. Instead, only those hedges should be quantified, which, for instance, occur two or more times.*
- ***Configuration:*** *All presented constraints must be configurable on the needs of a user. Rules should be assigned to member functions. Based on those rules, for instance, an expert system may derive optimal decisions.*

2) Instantiation: Inputs are a well-formed XTL template, as well as one or more instantiation sources. Further formal and non-formal requirements not mentioned in sect.III-A1 are:

1) *The implementation is to be done in Haskell. The Haskell-Toolbox for XML-processing [58] shall be used (see sect.V).*
2) *The implementation should essentially not deviate from the denotational semantics.*
   The data model and rules should be simple. Invalid XTL-tags should be treated as atomic element nodes.

3) Validation: XTL-conform schemas and XML instance document count as validation input. The result of validation is a "'yes"'/"'no"' answer. The requirements to a validator are as following:

1) *Definition of an appropriate data model.*
2) *Rules have to be minimal w.r.t. amount and length.*
   The premise of each matching case contains one node for the schema and one node, for instance. The validation of any node from the schema may only refer to the actual corresponding instance node, references to the following nodes are not allowed.
   The next rule to be applied shall be non-deterministic. No additional assertions on the selection are allowed. This approach is similar to instantiation.
3) *Before a validation fails, all other alternatives need to be checked first.*

   The error message should trace the actual error location and reason. If validation succeeds, a console notification should be emitted.

4) Unification: One main goal of unification of XTL-instantiation and validation is the lingual unification of both processes (cf. sect.I). Apart from that, the reuse of templates as schema shall be examined.

A rule-based approach would be desired for a better understanding and a qualitative investigation (see sect.VI). Here agreed data models should be used for both processes. Helper functions should be reused as much as possible. It requires those functions to be as generic as possible. Predefined functions shall be reused (cf. [21]).

5) Implementation in Java: The programs to be written in Haskell shall later be implemented in Java. Therefore, at least an object-oriented design and a translation of the denotational semantics into Java are needed (see sect.II). Here several questions emerge:

1) *How is polymorphism [22] and functionals implemented accordingly in Java?*
2) *Can the non-deterministic top-down automata remain as is?*
3) *What are appropriate class candidates? What do associations look like between them?*
4) *Which roles can be abstracted, and how do these roles interact?*
5) *How will most generic implementations look?*
6) *How do the architectural design patterns look?*

In this work, two programs are implemented, one for instantiation and one for validation. A test suite is introduced. Existing XSD schemas guarantee the well-formedness and validity of XTL-templates. Existing frameworks for XML processing are being used where appropriate.

IV. Design

In this section, introduced data models and semantics for instantiation and validation are presented.

Haskell is used as a programming language. Haskell's functional character allows a straight transformation from denotational semantics (cf. sect.I).

A. Data models

The goal of data models introduced in this section is easy denotational semantics.

The features set of HXT is relatively tiny. Usability is medium – so compromises must be made. So, the simpler the description gets, the more valuable the simplification becomes. In conclusion, the need arises to transform models. It takes small efforts instead when it comes to validation since transformation back again is not needed. In contrast to this, instantiation requires both transformation directions.

Furthermore, the data model transformation completeness and correctness need to be assured by covering both domains and co-domains. Those coverings may be used as a test suite for the implementations.

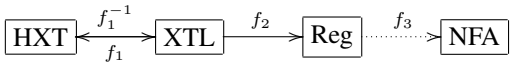


Fig. 3. Data models for Instantiation and Validation

Fig.3 shows the transformation for all the data models considered in this section. The data models `HXT` *and* `XTL` *represent document nodes. These nodes can be transformed by the functions f_1 and its inverse f_1^{-1} into each other. Both found the formal semantics for instantiation and validation. The composed mapping* `HXT` $\mapsto$ `XTL` $\mapsto$ `HXT` *describes instantiation. The first transformation transfers a parsed* `XmlTree` *into a simple fine-grained XTL-representation.*

In conclusion, the instantiated XTL document is transformed back to `HXT` *(see sect.IV-B). In addition to that, validation turns both XTL representations, for schema and instance, into the* `Reg`*-model representing regular expressions (see*

sect.IV-C1). A transformation back to `HXT` is not needed.

The models `HXT`, `XTL` and `Reg` are concretisations. All these models have equivalent expressibility, but each model has unique elements characteristic only for those. The `HXT`-model is only partially considered. All concretisations obey the following strict ordering:

$$HXT \ll XTL \ll Reg$$

The model `NFA` is only sketched. It is only to be an alternative proposition. Besides the disadvantages of sect.IV-A4 NFAs are not compatible by default with hedge-based data models like `HXT` and `XTL`. That is because NFAs are hard to represent as a tree. After all they are graphs in general.

The coverage of the domain determines the completeness of a model transformation. Coverage is considered separately in each section.

1) HXT: The denotational semantic uses functions from the Haskell XML Toolkit [58], solely for input and output processing of XML documents (see sect.V). The contained data structure '`XmlTree`' is the foundation for further processing using the toolkit. An XML-node '`XmlTree`' and a hedge '`XmlTrees`' are defined in Haskell as follows:

```
type XmlTree  = NTree XNode
     XmlTrees = [XmlTree] .
```

The type constructor '`NTree`' represents multi-way trees in general and is defined in GHC as

```
type NTree a = NTree a [NTree a] .
```

Here, the node type '`XmlTree`' explicitly denotes '`XNode`', so child nodes may only be one of these:

```
XText String
| XAttr QName
| XTag QName XmlTrees
```

'`QName`' denotes a qualified name. It may occur in element nodes. In HXT, these are composed of the type constructor '`QN`', a namespace prefix, a local identifier and a URI. The ordering is defined as:

```
type QName = QN ns local uri
```

So, the XML-node `<a id="1"/>` is represented as `XmlTree` as following:

```
NTree (XTag (QN "" "a" "")
  [NTree (XAttr (QN "" "id" "")
    [NTree (XText "1") []])])
```

The disadvantages of HXT are obvious. Even simple '`XmlTrees`' are very long and heavily loaded with brackets. So, it would be quite hard to experience the benefits of pattern-matching here. Since a clear and simple denotational semantic of a node is a precondition for simple processing semantic in general, composed combinators (see sect.III) are not equivalent.

Although the usability of the '`XmlTree`'-model in HXT is difficult at least, the amount of features in HXT is quite big (see sect.V).

Multiple functions and constructs overlay and are usable in some special cases only.

Implicit assumptions often cannot be seen by a function's name. For example, the type constructor '`XAttr`' and the constructor function '`xattr`' can build up attribute nodes. '`xattr`' implicitly insists attributes are specified first for children node constructions. This circumstance and unintentional constructor errors cause tree constructions to become hard to read and bloated quickly. Another issue is that all data types, type definitions and functions are loaded into their environment. The loading takes place as soon as an HXT-module is imported. A restricted module import command can resolve this problem. However, it requires a high level of awareness and can become very easy uncontrollable even with few imports.

Another severe problem is the too lax syntax of an XML node. So, many syntactic correct nodes may be generated which, however, are semantically not sound. Semantic mistakes may only be detected while serialising a document, only by throwing an exception. Otherwise, they will remain unnoticed. For example, attributes could accidentally be mistaken for element nodes because they have the '`XNode`' too. Also, the definition of a node in HXT does not prohibit an element node as an attribute node. The localisation of non-matching functions for that reason is a significant flaw. Often errors may only be localised manually by analysing the call stack. However, this is not sufficient. Mainly, due to a lack of good tool support for Haskell despite current attempts (cf. for instance with [29], [3]), the motivation rises even more to make data models and function as simple as possible.

2) XTL: The algebraic data type `XTL` is defined as

```
data XTL = XAtt String String
 | XTxt String
 | XInclude String
 | XMacro String [XTL]
 | XCallMacro String
 | XIf String [XTL]
 | XForEach String [XTL]
 | ElX String [(String,String)] [XTL]
 | TxtX String
```

Here, '`XAtt`' defines an attribute entry consisting of name and value. '`XTxt`' denotes an XTL text node (see sect.II-C). The type constructors '`XTxt`', '`XInclude`', '`XIf`', '`XForEach`' have '`select`'-expression as its first `String`. '`XMacro`', '`XIf`', '`XForEach`' denote a macro definition, condition and cycle. All of those have an '`ElX`' with `[XTL]` as a hedge. '`TxtX`' denotes an arbitrary XML text node. '`ElX`' is an XML node composed of a name, a list of attribute entries, and a child node hedge.

The mapping of HXT-nodes also affects XTL-tags (see sect.II), element and text nodes (see tab.IV). Element nodes and XTL-tags differ, for instance, when control should (not) depend on data. Comments and Processing-Instruction-nodes are not considered. XTL-tags not matching with command-tags should match with rule (El) and therefore should be transformed into usual element nodes. '`children2`' on the right-

$$(@)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"attribute"}\ \ \_\\ n = \mathrm{NTree}\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"name"}\ \ \texttt{""}))\\ [\mathrm{NTree}\ (\mathrm{XText}\ name)\ []]\\ s = \mathrm{NTree}\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"select"}\ \ \texttt{""}))\\ [\mathrm{NTree}\ (\mathrm{XText}\ select)\ []]\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [n, s])\ []\end{array}}{\mathrm{XAtt}\ name\ select}$$

$$(\#)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"text"}\ \ \_\\ s = \mathrm{NTree}\ (\mathrm{XText}\ select)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\mathrm{NTree}\\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"select"}\ \ \texttt{""}))\ [s]])\\ []\end{array}}{\mathrm{XTxt}\ select}$$

$$(I)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"include"}\ \ \_\\ s = \mathrm{NTree}\ (\mathrm{XText}\ select)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\\ \mathrm{NTree}\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"select"}\ \ \texttt{""}))\ [s]])\ []\end{array}}{\mathrm{XInclude}\ select}$$

$$(M)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"macro"}\ \ \_\\ t = \mathrm{NTree}\ (\mathrm{XText}\ mname)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\mathrm{NTree}\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\\ \texttt{"name"}\ \ \texttt{""}))\ [t]])\ children\end{array}}{\mathrm{XMacro}\ mname\ children2}$$

$$(C)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"callmacro"}\ \ \_\\ t = \mathrm{NTree}\ (\mathrm{XText}\ mname)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\mathrm{NTree}\\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"name"}\ \ \texttt{""}))\ [t]])\ []\end{array}}{\mathrm{XCallMacro}\ mname}$$

$$(If)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"if"}\ \ \_\\ t = \mathrm{NTree}\ (\mathrm{XText}\ select)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\mathrm{NTree}\\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"select"}\ \ \texttt{""}))\ [t]])\ children\end{array}}{\mathrm{XIf}\ select\ children2}$$

$$(FE)\ \frac{\begin{array}{l}a = \mathrm{QN}\ \texttt{"xtl"}\ \ \texttt{"for-each"}\ \ \_\\ t = \mathrm{NTree}\ (\mathrm{XText}\ select)\ []\\ \mathrm{NTree}\ (\mathrm{XTag}\ a\ [\mathrm{NTree}\\ (\mathrm{XAttr}\ (\mathrm{QN}\ \texttt{""}\ \ \texttt{"select"}\ \ \texttt{""}))\ [t]])\ children\end{array}}{\mathrm{XForEach}\ select\ children2}$$

$$(El)\ \frac{\mathrm{NTree}\ (\mathrm{XTag}\ qn\ atts)\ children}{\mathrm{ElX}\ qn2\ atts2\ children2}$$

$$(Txt)\ \frac{\mathrm{NTree}\ (\mathrm{XText}\ text)\ []}{\mathrm{TxtX}\ text}$$

TABLE IV
MAPPING HXT $\mapsto$ XTL

hand side of the rules (M), (If), (FE) and (El) denotes recursive continuation of the mapping onto the hedge ‘children’. *qn2 denotes a qualified name as a string, which qn generates. Because of the restriction of the HXT-model, the mappings are not injective, but they are surjective because all elements of* ‘XTL’ *are covered. Therefore an inverse mapping exists, which recursively applied to* ‘children2’ *results in* ‘children’.

The node `<a id="1"><b/></a>` *is represented in XTL as:*

```
ElX "a"  [("id","1")] [ElX "b"  [] []]
```

The XTL data model is used for instantiation and validation. Although instances do not have command tags, it is beneficial for unified semantics to express instances by XTL. It may also be used as an automated schema generator (see sect.IV-A3) or as a schema parser (see sect.VI).

3) Reg: The regular data model ‘Reg’ *is defined in Haskell as shown in fig.4:*

```
data Reg = MacroR String
 | AttrR String String
 | TextR String
 | IncludeR String
 | ElR String [(String,String)] Reg
 | TxtR String
 | Epsilon
 | Or Reg Reg
 | Then Reg Reg
 | Star Reg
```

Fig. 4. Data model Reg

‘Reg’ *follows the model presented in sect.IV-A2.* ‘AttrR’, ‘TextR’, ‘IncludeR’, ‘TxtR’ *and* ‘ElR’ *represent literals. Although element nodes are recursive, each can still be considered literal, especially when they are empty hedges or processed. The type constructor* ‘AttrR’ *has the same structure as* ‘XAtt’, ‘TextR’ *the same as* ‘XTxt’ *and* ‘IncludeR’ *as* ‘XInclude’. ‘Epsilon’ *denotes the empty word,* ‘Or’ *denotes selection,* ‘Then’ *denotes concatenation and* ‘Star’ *denotes arbitrary repetition (cf. [63]).*

In contrast to sect.IV-A2, regular expressions may not contain arbitrary macro calls. Otherwise, this could be considered as a non-right congruent derivation — this would be a context-free derivation. Regular expressions still can contain macro calls and can be unrolled initiated by a caller. In the following only those macros shall be considered whose derivation terminates. This statement means whose expression is regular. Replacement of regular expressions by other regular expressions preserves regularity according to the definition of tree grammars (see sect.II-D2).

The node

```
<a id="1">
 <xtl:attribute name="title"
   select="//AAA" />
 <b/>
</a>
```

matches ‘Reg’*:*

```
ElR "a" [("id","1")]
    Then (AttrR "title" "//AAA")
         (Then (ElR "b" [] Epsilon) Epsilon)
```

The introduced regular expressions can be interpreted as OBDD. OBDDs are graphical representations of terms consisting of variables and terms again, constants and binary descriptors. When extending the graphical notation proposed in fig.5 by the unary functors ‘Star’ *and* ‘e’*, an OBDD's representation is fully covered together with* ‘Then’, ‘Or’ *and* ‘@’*. Element nodes have a non-empty empty. Element nodes are already tree-structured and therefore can also be interpreted as nodes in OBDDs. Childless element nodes are leaves in a tree. This interpretation is isomorphic to boolean*

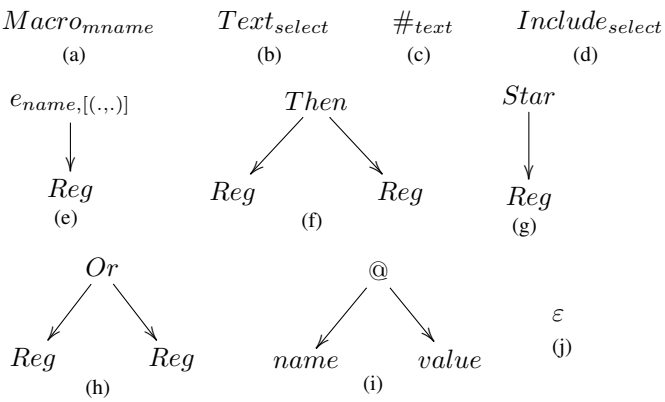


Fig. 5. Graphical Representation of `Reg`-nodes

terms. '$\wedge$' and '$\vee$' represent the binary functors, and boolean variables represent leaves.

The consideration of OBDDs has mainly two advantages here. First, the difference between nodes and hedge vanishes. A node representing a hedge with precisely one child is represented the same as a hedge with multiple children by `Then`*. Second, the set of alternatives is represented the same way (cf. [63], [17]). One advantage of a binary tree over a multi-way tree is a simpler specification of nodes.*

Furthermore, the graphical notation presented makes functions safer because fewer exceptions and cases need to be distinguished, and in conclusion there are fewer places to commit an error by the developer. There is either an '`Epsilon`*' or a '*`Then`*', where it is agreed '*`Then`*' may not have an '*`Epsilon`*' as its left child. Same as lists, the OBDD-notation allows lazy evaluation, so infinite OBDDs are a meaningful completion to the test cases from sect.V.*

A set of alternatives $\{a_0, ..., a_n\}$ can within a '`Then`*' be arranged in different ways (see fig.6). For instance, the '*`XTL`*'-node:*

```
ElX "a"  [("id","1")] [ElX "b"  [] []]
```

is transferred into the '`Reg`*'-node:*

```
ElR "a"  [("id","1")] Then (ElR "b"  []
                Epsilon) Epsilon
```

The composed node of XTL-command tags:

```
<book>
 <title>Haskell</title>
  <xtl:for-each select="//authors">
   <author>
     <xtl:text select="."/>
   </author>
 </xtl:for-each>
</book>
```

is transformed into a '`Reg`*':*

```
ElR "book" []
  Then
    (ElR "title" []
      Then (TxtR "Haskell") Epsilon)
    Then
      (Star
        (ElR "author"
         Then (TextR ".") Epsilon))
      Epsilon
```

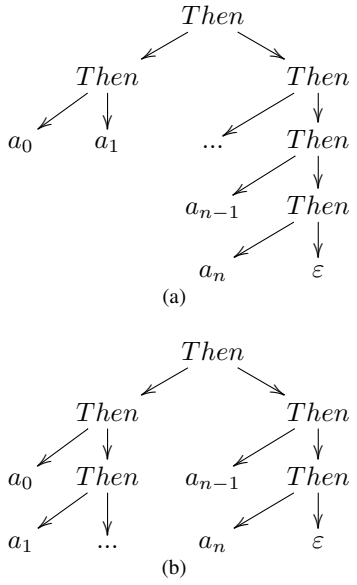


Fig. 6. Two different orderings of `Then`

The normalisation of regular expressions can simplify the validation. Arbitrary orderings of '`Then`*' and '*`Or`*' are disallowed, significantly reducing the number of rules to be considered. It is agreed that a regular expression is in normal form, if:*

- *No two text nodes are neighbours. Text inclusions are exempted.*
  Strings can be separated only with difficulties. It is because it is hard to decide with the truncation of a string is correct. The intention of two '`xtl:text`*' could be the merge into one given string. That is why a truncation may not make sense.*
- *'*`Then`*' and '*`Or`*' are right-associative.*
  It implies that there is a regular expression on both the left and the right, and the left expression does not match with the parent node
- *'*`XAtt`*' directly underneath a '*`Then`*' as the first contiguous sequence in the corresponding hedge '*`ElR`*' (cf. [32]).*

Using OBDDs qualitative properties may be probed (see sect.V, cf. [15]) because fewer base cases require consideration.

The mapping `XTL` $\mapsto$ `Reg` is total. Each '`XTL`*'-element is injectively assigned to an element from '*`Reg`*'. If '*`select`*'-expressions are ignored (see sect.III) and arbitrary non-fixed*

repetitions can be introduced using Kleene's star operator. The mapping is from now on, then no more invertible, s.t. the origin is reproducible exactly. 'XMacro' can be mapped onto empty since it is just added to a macro environment, and no real regular expression is associated with it.

XIf _l	↦	Or Epsilon l2
XForEach _l	↦	Star l2
XAtt name value	↦	AttrR name value
XTxt select	↦	TextR select
XInclude select	↦	IncludeR select
TxtX text	↦	TxtR text
ElX name atts l	↦	ElR name atts l2
XCallMacro mname	↦	MacroR mname

TABLE V
MAPPING XTL ↦ REG

The exact mapping is in tab.V.

'l2' denotes hedges recursively generated by the hedge 'l'. The inverse mapping from 'Reg' onto 'XTL' is not possible, not even by introducing "'don't-care"' variables '_' or referring to implicit macro environments. However, validation does not insist on it.

4) Non-deterministic Finite Automaton: As already mentioned in sect.II-D, regular tree automata just perfectly fit when it comes to recognising tree-structured regular input data, for instance, as XML documents are (cf. [50]). The construction of a Non-deterministic Finite Automaton (NFA) over hedges is similar to that over strings. It follows the so-called "'toolbox"'-principle. This principle states all nodes are applied nodes successively, for instance, during concatenation. After application, the end-points of one component are connected, so the resulting automaton grows (cf. [63]).

The partial-derivatives algorithm [9] derives only symbols lying in $\pi(t)$, where π denotes the set of all good beginnings, and t denotes an arbitrary yet to be determined regular expression. Due to the homomorphism for regular expressions (see sect.III-A4), the calculations can be placed within the remainder field modulo a literal. In the congruency $a \equiv b \bmod (c)$, a stands for an initial regular expression, b stands for the abstracted congruency reduced by c, and c is the remainder partition or, with other words, another possible beginning of a, so $c \in \pi(a)$.

The construction of the corresponding NFA always depends on the current derivation. All calculated derivations are put into a hashing table. Every time an expression is derived, it is first checked whether this derivation is already in the hashing table. For the original schema, $x^* \cdot (xx + y)^*$, either x (4.1) or y (4.3) can be derived. Due to the star-operator (4.1) can either have the same expression again or an x, because x^* would have been removed from $x^* \cdot (xx + y)^*$ and the second subexpression $(xx + y)^*$ x would follow after x, which, however, would follow another $(xx + y)^*$ (4.2). The congruencies (4.4)-(4.6) can be obtained after further derivations. Until (4.6) all right sides are determined. So, the corresponding NFA has no new transitions to be added (cf. app.X).

$$x^* \cdot (xx + y)^* \equiv x^* \cdot (xx + y)^* \quad \bmod (x) \quad (4.1)$$
$$x^* \cdot (xx + y)^* \equiv x \cdot (xx + y)^* \quad \bmod (x) \quad (4.2)$$
$$x^* \cdot (xx + y)^* \equiv (xx + y)^* \quad \bmod (y) \quad (4.3)$$
$$x \cdot (xx + y)^* \equiv (xx + y)^* \quad \bmod (x) \quad (4.4)$$
$$(xx + y)^* \equiv x \cdot (xx + y)^* \quad \bmod (x) \quad (4.5)$$
$$(xx + y)^* \equiv (xx + y)^* \quad \bmod (y) \quad (4.6)$$

Such an approach is appropriate for complex schemas, which may be reused (cf. sect.III). The state-based approach is also appropriate for error location. However, multiple transitions leaving a terminal and ε-transitions may make recognition not determined. Determination of the NFA (see [23]) comes for Rabin-Scott's powerset construction cost. Validation would be beneficial only if the automaton were determined. The additional cost pays off if multiple instances are validated with the same DFA.

On the one side, there is the direct approach as described in sect.IV-B2, IV-C1. On the other side, there is the graph-based approach. The validation problem can be interpreted as a path problem in a graph.

B. Instantiation

Before presenting the denotational semantics for instantiator and validator, a brief discussion on semantics should sum up the pros and cons.

1) Semantics Form: As already described in sect.III, the untyped λ-calculus and attribute grammar are not appropriate for describing the instantiation semantics.

A logical model seems reasonable at first glance since a matcher algorithm would be needed to be designed (cf. sect.III). A first implementation may even allow backtracking until an optimisation could be found.

The structure to be matched against can undoubtedly be represented as term expression (cf. [31]). Schema, instance, and helper functions can be transformed into Horn-clauses. Relations can only poorly express functions since logical programming mainly interprets terms and relations. Functions, however, have "'only"' static mappings. Apart from that, future implementations in an object-oriented programming language should guarantee referential transparency, and the evaluation ought to proceed forward. Concepts like backtracking and cuts in a logical programming language, however, disallow this — to mention some on Prolog, for instance.

For these reasons, the semantics of instantiation and validation should be applied to the functional paradigm (cf. [2], [61]).

The semantics shall be as easy as possible using type constructors, so a comprehending implementation in Haskell matches the denotational semantics. Node specifications using type constructors describe interfaces and attributes of classes (cf. sect.V-D).

Funktion	XTL-Konstruktor	Typung
f_0^1	`XTxt, XAtt`	`String→a→String`
f_0^2	`XForEach`	`String→a→[a]`
f_0^3	`XIf`	`String→a→Bool`
f_0^4	`XInclude`	`String→a→XmlTree`

TABLE VI
PLACEHOLDER-PLUGIN FUNCTIONS $f_0^1, f_0^2, f_0^3, f_0^4$

Line	Document	μ
0	...	$\emptyset$
1	`<xtl:macro name="x">`	$\{x/$`<a/>`$\}$
2	`  <a/>`	
3	`</xtl:macro>`	
4	`<xtl:macro name="y">`	$\{x/$`<a/>`$\}$
5	`  <b/><b/>`	
6	`</xtl:macro>`	$\{x/$`<a/>`$, y/$`<b/><b/>`$\}$

TABLE VII
MACRO ENVIRONMENT μ FOR AN EXAMPLE

Domain	Meaning
`Bool`	`{True, False}`
`XTL`	XTL data type
`[XTL]`	hedge of `XTL`

TABLE VIII
DOMAINS OF THE INSTANTIATION

2) Semantic: The complete semantics for instantiation and validation are enclosed in app.IX Instantiation turns a template into an instance. The instance document can be represented as XTL-term (cf. sect.IV-A2). The denotational semantics is described in Haskell. Next, some helper functions will be described in the denotational semantic:

The functions:

`filter::(a→Bool)→[a]→[a]` *and*
`concatMap::(a→[b])→[a]→[b]`

are defined in the GHC-package `GHC.List`*. The function '*`filter`*' filters any given list using a predicate (means a function returning a boolean value here). So, for instance,* `filter (odd) [1..10]` *returns as result the list* `[1,3,5,7,9]`*. The function '*`concatMap`*' applies for any given list each element in some given mapping, where for each list element, a list type is returned as an element of the co-domain. Later all created sub-lists are concatenated with each other.* `concatMap (\x->['a']) [1..10]` *returns* `"aaaaaaaaaa"`*.*

The function '`qSort`*' of type '*`Ord a⇒[a]→[a]`*' is a generalised (so-called lifted) function sorting a list of any comparable type. '*`qSort`*' is used for canonisation.*

The functions $f_0^1, f_0^2, f_0^3, f_0^4$ denote the agreed PHP functions (cf. sect.III). These functions describe in the given order access to external texts, the filtering mode if there are multiple solutions, the checking on satisfiability and the return of an element node. The corresponding types can be found in tab.VI. The type '`a`*' denotes arbitrary instantiation elements, for instance, element nodes for XML documents. Be aware of '*`XInclude`*'. It always produces an '*`XmlTree`*' to the instance document. The result of function f_0^2 is '*`[a]`*'. That means the polymorphic results are being passed through as context to the loop body.*

The helper function `getMacro` *of type '*`[XTL]→[XTL]`*' from (A3) returns for a macro environment μ the corresponding macro body. It is further assumed, a matching macro is defined before calling it. Otherwise, an according to error message shall be dumped. The surrounding rule by name* `m1` *transfers the name of the wanted macro definition.*

The abbreviation (S) stands for the starting rule of instantiation, (E) stands for eliminating XTL-attributes, which are located directly under the top-level element node. Rules (I1)-(I3) initiate instantiation by filtering all macro-definitions first and passing those to the instantiation second. Rules (A1)-(A7) are the core instantiation rules for XTL-tags and non-XTL-tags.

The helper function ε^{MA} of type '`XTL→Bool`*' checks for a given '*`XTL`*'-node if it is, in fact, an XTL-attribute or not. In analogy, ε^{MM} checks for macro-definitions. The semantic fields include macro-environment and context (see app.IX). Here, a macro environment μ is built up in (I3). μ consists of the mapping "'Macro-Name×XTL-Hedge"'. The hedge represents a list. The macro-environment μ is an invariant part in ε^{α} (see app.IX). Tab.VII demonstrates the macro environment's evaluation (cf. sect.II-C) until the selected row.*

Context has multiple purposes, one of which is a simplification. For instance, the simplification of XPath-expressions (see sect.II, III). It returns a node or a value of arbitrary type and is used while calling PHP functions as the instantiation data source.

The mappings use the domains from tab.VIII. '`XTL`*' describes the domains in more detail in sect.II-C. '*`[XTL]`*' denotes a hedge of type '*`XTL`*'. The hedge type is a list in terms of denotational semantic. It is worth noting the type is monadic.*

The considered denotational semantic variables are used quite often. Anonymous and "'don't-car"' variables are marked as '_'. These variables often stand in constructor specification and cannot be addressed in a rule's body. There are underlined variables written in italics – these specify XTL-nodes and fragments of it. These can also be memoised. For example, XTL-attributes in (E) can be interrupted and escaped before reaching (E)'s end. The same counts for non-XTL-attributes in `nodes` *too. Non-underlined variables with a Greek letter π and μ denote each PHP-tuple ($f_0^1, f_0^2, f_0^3, f_0^4$) and also the macro environment. Functions are written in italics, and each has an upper index, which denotes the arity of that function. The only exceptions are doubly-indexed PHP-function from f_0^1 to f_0^4. The arities of those functions are slightly different and are fully listed in tab.VI.*

Rules of the denotational semantic

The rule

(S) $\mathcal{E}^{Start}[\![\underline{x}]\!]_{\underline{s}}\pi := \mathcal{E}[\![\underline{x2}]\!]_{\underline{s}}\pi$ *for* $\underline{x2} = \mathcal{E}^{r}[\![\underline{x}]\!]$

initiates instantiation.

x denotes the template, *s* denotes the context, and π denotes the placeholder-function 4-tuple. The given template is a non-empty `XTL`-node. The context is the source document at the beginning. Because *x* may also have XTL-attribute in the hedge, *X* is reduced first of all with $\mathcal{E}^r[\![]\!]$.

(E) $\mathcal{E}^r[\![\texttt{ElX n a c}]\!] :=$
let attDefs = $\overline{\text{filter}}^2(\lambda\texttt{child}.\mathcal{E}^{MA}[\![\texttt{child}]\!])\texttt{c}$,
nodes = $\overline{\text{filter}}^2(\lambda\texttt{child}.\overline{\text{not}}^1\ \mathcal{E}^{MA}[\![\texttt{child}]\!])\texttt{c}$
in `ElX n` ($\overline{\text{qSort}}^1$ (a ++ attDefs)) nodes

It is implicitly assumed that the top-level root node is an element node that may not be an XTL-tag. The hedge is split into one hedge containing '`xtl:attribute`' *and another hedge containing all others.* attDefs *binds* '`xtl:attribute`'*-nodes.* nodes *bind all others. Then the original attributes* a *are united with* attDefs *and canonised last. This list and the remaining nodes* nodes *and the unmodified element name* n *make up the reduced element node. Afterwards, instantiation is triggered.*

The pure instantiation shall be implemented as filter (see sect.III-A5). It returns a list type. The triggering $\mathcal{E}[\![]\!]$ *requires one* '`XTL`' *only:*

(I1) $\mathcal{E}[\![\texttt{XTxt t}]\!]\_\ \_ :=$ `XTxt t`

(I2) $\mathcal{E}[\![\texttt{XAtt n v}]\!]\_\ \_ :=$ `XAtt n v`

(I3) $\mathcal{E}[\![\texttt{ElX n a c}]\!]s\pi :=$
let mdefs = $\overline{\text{filter}}^2(\lambda\texttt{child}.\mathcal{E}^{MM}[\![\texttt{child}]\!])\texttt{c}$,
nodes = $\overline{\text{filter}}^2(\lambda\texttt{child}.\overline{\text{not}}^1\ \mathcal{E}^{MM}[\![\texttt{child}]\!])\texttt{c}$
in `ElX n a` ($\overline{\text{concatMap}}^2$
(λnode.$\mathcal{E}^{\alpha}[\![\texttt{node}]\!](s, \texttt{mdefs}, \pi)$) nodes)

In both cases, (I1) and (I2) attributes and text nodes are considered. Case (I3) evaluates one element node, which may have occurrences of macro calls in the case of a hedge. These are filtered, similar to (E), and filtered afterwards to mdefs. *Nodes that are no macros are bound to* nodes. *Later the instantiation* $\mathcal{E}^{\alpha}[\![]\!]$ *proceeds for each non-macro node* node. $\overline{\text{concatMap}}^2$ *concatenates obtained hedges. The instantiation of each* node *is passed through referring to source* s *and PHP-tuple* π. *The list of all macros for this rule serves in* $\mathcal{E}^{\alpha}[\![]\!]$ *as macro environment* μ *and remains untouched during the remaining instantiation.*

The instantiation of '`XTL`'*-nodes has the typing:*

$$\mathcal{E}^{\alpha}[\![]\!] :: \texttt{XTL} \to \texttt{a} \to \texttt{[(String,[XTL])]}$$
$$\to \texttt{PHP a} \to \texttt{[XTL]}.$$

This means the hedge of type `[XTL]` *establishes, after an XTL-node, instantiation data of type* '`a`' *and a macro environment are passed. The macro-environment consists of a list of tuples herewith, where each tuple has the mapping "'Macro-Name↦Macro-Body"'. The macro body is a hedge of type* '`[XTL]`' *and may contain all XTL-tags except* '`XMacro`'.

Instantiation is described by rules (A1) until (A7). Notably, the tag '`XMacro`' *is missing. That is because the extraction of* μ *happens before. Furthermore, all seven rules are complete. W.r.t. XTL-tags have not imposed any further restrictions. The rules are not prioritised. Element nodes that are non-XTL-tags are handled as common element nodes in the semantic (cf. sect.IV-A2).*

The handling of conditions is in (A1).

(A1) $\mathcal{E}^{\alpha}[\![\texttt{XIf x gc}]\!](s, \mu, (\bar{f}^1_0, \bar{f}^2_0, \bar{f}^3_0, \bar{f}^4_0)) :=$
if ($\bar{f}^3_0$ xs) *then* $\overline{\text{concatMap}}^2$
(λc.$\mathcal{E}^{\alpha}[\![\texttt{c}]\!](s, \mu, (\bar{f}^1_0, \bar{f}^2_0, \bar{f}^3_0, \bar{f}^4_0))$)gc
else `[]`

So, the string x *is passed to the PHP-function* f^3_0 *together with* s, *the instantiation data source. If* f^3_0 *succeeds (cf. sect.III), it returns* '`True`', '`False`' *otherwise. The Haskell syntax implies that on success, instantiation continues with child node* c. *In case of error, it returns an empty list. However, an empty list is neutral w.r.t. list concatenation, and this is why no special handling is required on the caller's side.*

The handling of loops is done in (A2).

(A2) $\mathcal{E}^{\alpha}[\![\texttt{XForEach x gc}]\!](s, \mu, (\bar{f}^1_0, \bar{f}^2_0, \bar{f}^3_0, \bar{f}^4_0)) :=$
let sels = $\bar{f}^2_0$ x s
in $\overline{\text{concatMap}}^2$ ($\overline{f3}^1$) sels
for $\overline{f3}^1 = \lambda$c.$\overline{\text{concatMap}}^2$
(λc2.$\mathcal{E}^{\alpha}[\![\texttt{c2}]\!](\texttt{c}, \mu, (\bar{f}^1_0, \bar{f}^2_0, \bar{f}^3_0, \bar{f}^4_0))$)gc

The variable sels *binds all nodes of type* '`a`', *which establish during the evaluation of the* '`select`'*-expression* x, *the source of instantiation data* s, *and the PHP-function* f^2_0. *In case* sels *equals an empty list, then function* $\overline{\text{concatMap}}^2$ *results in the empty set. That is because Haskell has a non-strict evaluation order. Otherwise, first, instantiation continues for each child node* c2 *of hedge* gc, *and second, all determined hedges are then concatenated together into eventually one resulting hedge. The list* sels`::[a]` *is successively propagated through to* $\overline{\text{concatMap}}^2$ *using the bound variable* c. *Every element in* sels *has type* '`a`' *and is passed over to each child node* c2 *for hedge* gc *as new instantiation data* c.

The handling of macro calls is done in (A3).

(A3) $\mathcal{E}^{\alpha}[\![\texttt{XCallMacro m1}]\!](s, \mu, \pi) :=$
let μ_2 = $\overline{\text{getMacro}}^1\ \mu$
in $\overline{\text{concatMap}}^2(\lambda\texttt{m}.\mathcal{E}^{\alpha}[\![\texttt{m}]\!](s, \mu, \pi))\ \mu_2$

The function '`getMacro`' *was mentioned initially. It determines for a given macro name* m1 *and a macro-environment* μ *the corresponding macro body* μ_2, *a hedge. The instantiation proceeds for each node sequentially from the hedge with the same initial context* s, *PHP-tuple* π *and the same macro-environment* μ. *The resulting hedge as the list is concatenated, so this rule's returned value has the type* '`[XTL]`'.

The inclusion of text follows rule (A4).

(A4) $\mathcal{E}^{\alpha}[\![\texttt{XTxt t}]\!](s, \mu, (\bar{f}^1_0, \_, \_, \_)) :=$
`[ XTxt` $\bar{f}^1_0$ t s `]`

Access is granted by the function $\bar{f}^1_0$ *for each* '`select`'*-expressions* t *and instantiation data* s. *The instantiation result is a singular list of text nodes, which is determined by the evaluation. Access to XML-nodes of instantiation data is granted by rule (A5), which is analogous to (A4) except that*

f_0^4 *returns an XML-node.*

(A5) $\mathcal{E}^\alpha[\![\texttt{XInclude}\ \underline{x}]\!](\underline{s},\mu,(\_,\_,\_,\bar{f}_0^4)) :=$
$[\ \bar{f}_0^4\ \underline{x}\ \underline{s}\ ]$

Instantiation of common nodes is done by (A6)-(A7).

(A6) $\mathcal{E}^\alpha[\![\texttt{ElX}\ \underline{n}\ \underline{a}\ \underline{c}]\!](\underline{s},\mu,\pi) :=$
$[\ \texttt{ElX}\ \underline{n}\ \underline{a}$
$(\overline{\textit{concatMap}}^2(\lambda\underline{child}.\mathcal{E}^\alpha[\![\underline{child}]\!](\underline{s},\mu,\pi))\ \underline{c})\ ]$

(A7) $\mathcal{E}^\alpha[\![\texttt{TxtX}\ \underline{t}]\!](\_,\_,\_) :=$ `[ TxtX t ]`

Element nodes are taken as-is. Instantiation proceeds with children nodes with the same instantiation data, macro environment and PHP-tuple. Text nodes are just copied unconditionally.

Example
Let the following document be given

```
<books>
  <xtl:for-each select="//book">
    <title>
     <xtl:text select="@title"/>
    </title>
  </xtl:for-each>
</books>
```

where $\underline{x}$ *is the bibliography document wanted from sect.II-A.*

So, the following then holds

$\pi = (\bar{f}_0^1, \bar{f}_0^2, \bar{f}_0^3, \bar{f}_0^4)$
$\underline{s} =$ `ElX "books" []`
`[XForEach "//book" [ElX "title"`
`[] [XTxt "@title"]]]`

where

$\underline{sel1} =$ `ElX "book" [("author","Simon`
`..."),("title","Has...")] []`
$\underline{sel2} =$ `ElX "book" [("author","Joshua`
`..."),("title","Re...")] []`

denote element nodes from $\underline{s}$*. These are reused as following: The derivation during instantiation starts with* $\mathcal{E}^{Start}[\![]\!]$ *and is interrupted on the first occurrence of '...' by* $\mathcal{E}^r[\![]\!]$*. The remaining segments are composed in analogy to that and are also disrupted by minor calculations.*

$\mathcal{E}^{Start}[\![\texttt{ElX "books" [] [...]}]\!]\underline{s}\pi$
$= \mathcal{E}[\![\underline{x2}]\!]\underline{s}\pi$ *for* $\underline{x2} = \mathcal{E}^r[\![\texttt{ElX "books" [] [...]}]\!]$
$= \ldots$
$= \mathcal{E}[\![\underline{x2}]\!]\underline{s}\pi$ *for* $\underline{x2} =$ `ElX "books" [] [XForEach`
`"//book" [...]]`
$= \mathcal{E}[\![\texttt{ElX "books" [] [...]}]\!]\underline{s}\pi$
$=$ ***let*** $\underline{mdefs} = \overline{\textit{filter}}^2(\lambda\underline{child}.$
$\mathcal{E}^{MM}[\![\underline{child}]\!])$`[XForEach "//book" [...]],`
$\underline{nodes} = \overline{\textit{filter}}^2(\lambda\underline{child}.\overline{\textit{not}}^1\ \mathcal{E}^{MM}[\![\underline{child}]\!])$
`[XForEach "//book" [...]]`
in `ElX "books" []` $(\overline{\textit{concatMap}}^2\ (\lambda\underline{node}.$
$\mathcal{E}^\alpha[\![\underline{node}]\!](\underline{s},\underline{mdefs},\pi))\ \underline{nodes})$
$=$ ***let*** $\underline{mdefs} =$ `[],`
$\underline{nodes} =$ `[XForEach "//book" [...]]`
in `ElX "books" []` $(\overline{\textit{concatMap}}^2(\lambda\underline{node}.$
$\mathcal{E}^\alpha[\![\underline{node}]\!](\underline{s},\underline{mdefs},\pi))\ \underline{nodes})$
$=$ `ElX "books" []`
$(\overline{\textit{concatMap}}^2(\lambda\underline{node}.\mathcal{E}^\alpha[\![\underline{node}]\!](\underline{s},$ `[],`$\pi))$
`[XForEach "//book" [...]])`
$=$ `ElX "books" []` $(\overline{\textit{concat}}^1$ `[`
$\mathcal{E}^\alpha[\![$`XForEach "//book" [...]`$]\!](\underline{s},$ `[],`$\pi)$`])`
$= \ldots$
$=$ `ElX "books" []` $(\overline{\textit{concat}}^1$ `[[`
`ElX "title" [] [TxtX "Has..."],`
`ElX "title" [] [TxtX "Re..."]]])`
$=$ `ElX "books" [ElX "title" [] [TxtX`
`"Has..."], ElX "title" [] [TxtX`
`"Re..."]]` ■

$\mathcal{E}^r[\![\texttt{ElX "books" [] [...]}]\!]$
$=$ ***let*** $\underline{attDefs} = \overline{\textit{filter}}^2(\lambda\underline{child}.\mathcal{E}^{MA}[\![\underline{child}]\!])$
`[XForEach "//book" [...]],`
$\underline{nodes} = \overline{\textit{filter}}^2(\lambda\underline{child}.\overline{\textit{not}}^1\mathcal{E}^{MA}[\![\underline{child}]\!])$
`[XForEach "//book" [...]]`
in `ElX "books"` $(\overline{\textit{qSort}}^1(\texttt{[]} +\!\!+ \underline{attDefs}))\ \underline{nodes}$
$=$ ***let*** $\underline{attDefs} =$ `[],`
$\underline{nodes} =$ `[XForEach "//book" [...]]`
in `ElX "books"` $(\overline{\textit{qSort}}^1(\texttt{[]} +\!\!+ \underline{attDefs}))$
$\underline{nodes}$
$=$ `ElX "books" [] [XForEach "//book"`
`[...]]` ■

$\mathcal{E}^\alpha[\![\texttt{ElX "title" [] [...]}]\!](\underline{sel1},$ `[],`$\pi)$
$=$ `[ElX "title" []` $(\overline{\textit{concatMap}}^2(\lambda\underline{child}.$
$\mathcal{E}^\alpha[\![\underline{child}]\!](\underline{sel1},$ `[],`$\pi))$ `[XTxt "@title"]]`
$=$ `[ElX "title" []` $(\overline{\textit{concat}}^1$
$[\mathcal{E}^\alpha[\![$`XTxt "@title"`$]\!](\underline{sel1},$ `[],`$\pi)])]$
$=$ `[ElX "title" []` $(\overline{\textit{concat}}^1$ `[[TxtX` $\bar{f}_0^1$
`"@title"` $\underline{sel1}$`]])]`
$=$ `[ElX "title" []` $(\overline{\textit{concat}}^1$ `[[TxtX`
`"Has..."]])]`
$=$ `[ElX "title" [] [TxtX "Has..."]]` ■

$\mathcal{E}^\alpha[\![$`XForEach "//book" [...]`$]\!](\underline{s}, [], \pi)$
= ***let*** $\underline{sels} = \overline{f_0}^2$ `"//book"` $\underline{s}$
in $\overline{concatMap}^2(\overline{f3}^1)\ \underline{sels}$
for $\overline{f3}^1 = \lambda\underline{c}.\overline{concatMap}^2(\lambda\underline{c2}.\mathcal{E}^\alpha[\![\underline{c2}]\!](\underline{c}, [], \pi))$
`[ElX "title" [] [...]]`
= ***let*** $\underline{sels}$ = $[\underline{sel1}, \underline{sel2}]$
in $\overline{concatMap}^2(\overline{f3}^1)\ \underline{sels}$
for $\overline{f3}^1 = \lambda\underline{c}.\overline{concatMap}^2(\lambda\underline{c2}.\mathcal{E}^\alpha[\![\underline{c2}]\!](\underline{c}, [], \pi))$
$[$`ElX "title" []`$_2$ `[...]`$]$
= $\overline{concat}^1$ $[$ $\overline{concatMap}^2(\lambda\underline{c2}.\mathcal{E}^\alpha[\![\underline{c2}]\!](\underline{sel1}, [], \pi))$
`[ElX "title" [] [...]],`
$\overline{concatMap}^2(\lambda\underline{c2}.\mathcal{E}^\alpha[\![\underline{c2}]\!](\underline{sel2}, [], \pi))$
`[ElX "title" [] [...]] ]`
= $\overline{concat}^1$ $[$ $\overline{concat}^1$ $[\mathcal{E}^\alpha[\![$`ElX "title" [] [..]`$]\!]$
$(\underline{sel1}, [], \pi)]$,
$\overline{concat}^1$ $[\mathcal{E}^\alpha[\![$`ElX "title" [] [..]`$]\!]$
$(\underline{sel2}, [], \pi)]$ $]$
= $\overline{concat}^1$ $[$ $\mathcal{E}^\alpha[\![$`ElX "title" [] [...]`$]\!]$
$(\underline{sel1}, [], \pi)$,
$\mathcal{E}^\alpha[\![$`ElX "title" [] [...]`$]\!]$
$(\underline{sel2}, [], \pi)$ $]$
= ...
= $\overline{concat}^1$ $[$ `[ElX "title" [] [TxtX "Has..."]],`
`[ElX "title" [] [TxtX "Re..."]] ]`
= `[ElX "title" [] [TxtX "Has..."],`
`ElX "title" [] [TxtX "Re..."] ]` ■

$\mathcal{E}^\alpha[\![$`ElX "title" [] [...]`$]\!](\underline{sel1}, [], \pi)$
= `[ElX "title" []` $(\overline{concatMap}^2(\lambda\underline{child}.$
$\mathcal{E}^\alpha[\![\underline{child}]\!](\underline{sel1}, [], \pi))$ `[XTxt "@title"])]`
= `[ElX "title" []` $(\overline{concat}^1$
`[`$\mathcal{E}^\alpha[\![$`XTxt "@title"`$]\!](\underline{sel1}, [], \pi)$`])]`
= `[ElX "title" []` $(\overline{concat}^1$
`[[TxtX` $\overline{f_0}^1$ `"@title"` $\underline{sel1}$`]])]`
= `[ElX "title" []` $(\overline{concat}^1$ `[[TxtX`
`"Has..."]])]`
= `[ElX "title" [] [TxtX "Has..."]]` ■
$\mathcal{E}^\alpha[\![$`ElX "title" [] [...]`$]\!](\underline{sel2}, [], \pi)$
= `[ElX "title" []`
$(\overline{concatMap}^2(\lambda\underline{child}.\mathcal{E}^\alpha[\![\underline{child}]\!](\underline{sel2}, [], \pi))$
`[XTxt "@title"])]`
= `[ElX "title" []` $(\overline{concat}^1$
`[`$\mathcal{E}^\alpha[\![$`XTxt "@title"`$]\!](\underline{sel2}, [], \pi)$`])]`
= `[ElX "title" []` $(\overline{concat}^1$
`[[TxtX` $\overline{f_0}^1$ `"@title"` $\underline{sel2}$`]])]`
= `[ElX "title" []` $(\overline{concat}^1$
`[[TxtX "Re..."]])]`
= `[ElX "title" [] [TxtX "Re..."]]` ■

C. Validation

Schema and instance are represented as regular expressions on validation (see sect.IV-A3). Instances, however, do not have alternatives, no star-operator, no XTL attributes and no text inclusions – in contrast to schemas. Instances can be described neatly by ε and '`Then`'*. In the normal form, the last node of a recursive node in an OBDD ε is used.* '`Then`' *is recommended for node concatenation to a hedge.*

It means a matcher (see sect.III) must recognise the cases from fig.7. The left side denotes the set of instance nodes, and the right side denotes the set of schema nodes. All 32 cases of the bipartite graph need to be handled because XML instances a priori do not contain XTL-tags (cf. [32])

XTL is well-formed and safe according to sect.III. Except for bypasses and node inclusion, there is no other way to generate element nodes. A matching of hedges with node inclusion is no longer considered here because the essential question researched here is w.l.o.g. if two generated languages are the same or not – and for that, node inclusion does not matter in practice. An equality solver is needed, which would determine solutions just from the relations to address this issue. Here, a solution does not necessarily have to exist (see [23]). So, it is guaranteed that element nodes of the instance match only with tags that are not XTL-commands – meaning only element nodes.

1) Semantic: The validation bases on regular expressions '`Reg`'*, which are mapped onto boolean values $\mathbb{B}$. Interpretations of $\mathbb{B}$ map onto $\{$`True, False`$\}$. All XML instances and XTL-schemas are definable over* '`Reg`' *(see sect.IV-A3). Macros contain a head determined by a macro name and a macro body, a hedge of element nodes. The hedge is represented by a top-level* '`Then`' *and which is an OBDD. The information about which macro name is assigned which macro body must be available on validation as context information μ. The macro-environment μ has the typing* `String`$\rightarrow$`Reg`*.*

The following semantic-rules variables will be used differently, namely for the specification of instance and schema nodes and or strings. That allows a shorter rule representation than without, and by memoisation, it avoids redundant calculations.

The validation is done by

$$\mathcal{E}[\![\mathbb{I}, \mathbb{S}]\!] :: \texttt{Reg} \rightarrow \texttt{Reg} \rightarrow \texttt{Bool}$$

, where $\mathbb{I}$ denotes the instance and $\mathbb{S}$ the schema document.

Used helper functions are listed in tab.IX. The function '`qSort`' *is a lifted function of type* `[a]`$\rightarrow$`[a]` *and is used, particularly, for canonisation. The functions* $\overline{frontSplits}^1$ *and* $\overline{splits}^1$ *divide non-deterministically a regular expression into two disjoint parts. All parts of a* '`Reg`' *are calculated lazily. In contrast to* $\overline{splits}^1$, $\overline{frontSplits}^1$ *calculates one partition less — it skips the trivial partition* `(Epsilon,Reg)`*. The partition is considered here as an inversion of the* '`Then`'*-concatenation. It works in analogy to the non-deterministic partition of strings in* $\overline{frontSplitText}^1$ *and* $\overline{splitText}^1$.

Function		Type
frontSplits	::	Reg→[(Reg,Reg)]
splits	::	Reg→[(Reg,Reg)]
qSort	::	[(String,String)] →[(String,String)]
frontSplitText	::	String→[(String,String)]
splitText	::	String→[(String,String)]
extractAttributes	::	Reg→Reg
getMacro	::	String→[(String,Reg)]→Reg

TABLE IX
VALIDATION – HELPER FUNCTIONS

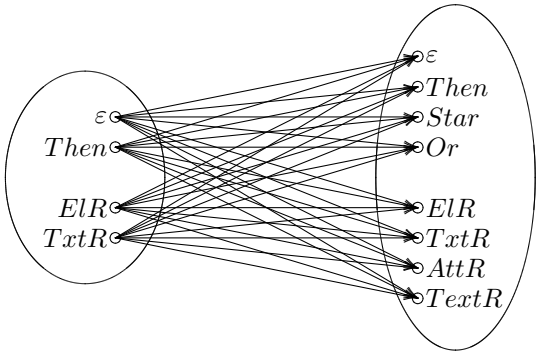


Fig. 7. Interleaving of cases

For instance, $\overline{splitText}^{1}$ `"ab"` *returns the list* `[("","ab"),("a","b"),("ab","")]`*, but* $\overline{frontSplitText}^{1}$ `"ab"` *only returns the last two tuples.*

$\overline{extractAttributes}^{1}$ *extracts from an element node given by a* '`Reg`' *all attributes* '`AttrR`' *and adds to preexisting ones into the element node. The function* $\overline{getMacro}^{2}$ *scans* μ *and find for a given macro name the matching body, a* '`Reg`'*. It is assumed during validation that all macros are defined before running it.*

The validation of a hedge is successful if all children of the schema successively match – not necessarily with the same position index. It is implied by the logical operator '$\wedge$*'. Alternatives are evaluated by '*$\vee$*'.*

The rules of the denotational semantic

The rules of the denotational semantic obey fig.7. Validation rules are not going to be explained just sequentially to illustrate the topic.

First of all, rules do not follow any order a priori here. However, it is still agreed upon the prioritisation of listed rules. The precedence of interleaving rules raises with the rule number. The following rules can so be described shorter.

Moreover, it is agreed upon in the matching relation $\cong$ *(see sect.III-A3) first argument in is the instance* $\mathbb{I}$ *to the left, and second argument the schema* $\mathbb{S}$ *to the right. Hence,* $\cong$ *as denotational semantic can be divided into four partitions according to the* '`Reg`'*-structure of the instance.*

(E1) $\mathcal{E}[\![$`Epsilon, TxtR ""`$]\!]\mu$:= `True`
(E2) $\mathcal{E}[\![$`Epsilon, TxtR _`$]\!]\mu$:= `False`
(E3) $\mathcal{E}[\![$`Epsilon, Epsilon`$]\!]\mu$:= `True`
(E4) $\mathcal{E}[\![$`Epsilon, ElR _ _ _`$]\!]\mu$:= `False`
(E5) $\mathcal{E}[\![$`Epsilon, Star _`$]\!]\mu$:= `True`
(E6) $\mathcal{E}[\![$`Epsilon, TextR _`$]\!]\mu$:= `True`
(E7) $\mathcal{E}[\![$`Epsilon, Then r1 r2`$]\!]\mu :=$
$\mathcal{E}[\![$`Epsilon, r1`$]\!]\mu \wedge \mathcal{E}[\![$`Epsilon, r2`$]\!]\mu$

In the first paragraph, empty instance nodes match with empty text nodes (E1) and (E6), and empty schema nodes (E3) matches with star-operator (E5). Exemptions apply to element nodes (E4) and non-empty strings (E2). The rules (E2) and (E1) overlap because (E1) is a particular case of (E2). However, the relatively simple rule (E2) is preferred over an explicit representation. That is why empty text nodes shall first match with (E1). (E7) considers the case a schema has a concatenation, whose left '`Then`'*-branch is not* ε *– but this node could still be derived to* ε*. That is why both branches of the schema-node must be derivable to* ε*.*

The second passage treats concatenations of instance nodes — this means hedges. Since the normal form of OBDDs in instances excludes two consecutive ε *and text nodes (see sect.IV-A3), and since the left branch of a* '`Then`' *may not be empty, it can be inferred an instance-hedge is not derivable to* ε *(Then1).*

(Then 1) $\mathcal{E}[\![$`Then _ _, Epsilon`$]\!]\mu$:= `False`

Moreover, the normal form implies that a hedge either entirely contains a string in the left branch and the right branch validates against ε*, or no validation is valid here (Then2). The derivation of the right branch is needed because the left branch does not have to be syntactically identical – it could also result from a hedge evaluation that requires attention.*

(Then 2) $\mathcal{E}[\![$`Then r1 r2, TxtR text`$]\!]\mu :=$
$\mathcal{E}[\![$`r1, TxtR text`$]\!]\mu \wedge \mathcal{E}[\![$`r2, Epsilon`$]\!]\mu$

This rule differs from

(Then 8) $\mathcal{E}[\![$`Then (TxtR _) Epsilon, TextR _`$]\!]\mu$:= `True`

(Then 9) $\mathcal{E}[\![$`Then _ _, TextR _`$]\!]\mu$:= `False`

An '`xtl:text`''*s instantiaion insists a text node in the instance document follows. There is no other validation (Then9).*

The validation of a hedge with the element node at the beginning (Then3) can only be successful against an element node in the schema if both element nodes are homomorphic w.r.t. validation and all remaining nodes of the hedge validate against ε*. All other cases lead to an unsound instance (Then4).*

(Then 3) $\mathcal{E}[\![$`X_L, ElR name2 atts2 r2`$]\!]\mu :=$
$X_L =$ `Then (ElR name1 atts1 r1) r`
$\mathcal{E}[\![$`ElR name1 atts1 r1, `$X_R]\!]\mu \wedge$
$\mathcal{E}[\![$`r, Epsilon`$]\!]\mu$
$X_R =$ `ElR name2 atts2 r2`

(Then 4) $\mathcal{E}[\![$`Then _ _, ElR _ _ _`$]\!]\mu$:= `False`

Rule (Then5) attempts to match a repeating sequence from a hedge non-deterministically. Here the given sequence is divided, s.t. the first part with hedge from the cycle, and the right part matches with the whole cycle, which may also be empty. If there is a repetition in the given hedge, then the

right ‘Then’*-branch may match the remaining hedge. One corner case occurs when the left* ‘Then’*-branch contains the whole cycle, so the remaining hedge* ε *successfully matches with* `Star s`.

(Then 5) $\mathcal{E}[\![$ `Then r1 r2, Star s` $]\!]\mu :=$
$\vee$ `[True|`
`(s1,s2)` $\leftarrow \overline{\textit{frontSplits}}^{1}$ `(Then r1 r2)`
$\wedge\ \mathcal{E}[\![$ `s1, s` $]\!]\mu \wedge \mathcal{E}[\![$ `s2, Star s` $]\!]\mu$ `]`

(Then 6) represents the case with a single child node, which according to the normalform occurs instead of a literal. In this case, for example, there cannot exist an element node in `s2`. *Generally considered, the left branch* `r1` *must match with the whole hedge from* `s` *and* `s2`. *This case is the base case for (Then 7), then (Then 7) initiates validation of both branches — which without (Then 6) would not necessarily terminate if the right branch derives to* ε.

(Then 6) $\mathcal{E}[\![$ `Then r1 Epsilon, Then s1 s2` $]\!]\mu :=$
$\mathcal{E}[\![$ `r1, Then s1 s2` $]\!]\mu$

(Then 7) $\mathcal{E}[\![$ `Then r1 r2, Then s1 s2` $]\!]\mu :=$
$\vee$ `[True|`
`(t1,t2)` $\leftarrow \overline{\textit{splits}}^{1}$ `(Then r1 r2)`
$\wedge\ \mathcal{E}[\![$ `t1, s1` $]\!]\mu \wedge \mathcal{E}[\![$ `t2, s2` $]\!]\mu$ `]`

The third passage considers text nodes. Empty text nodes are derivable to ε *(#2). Non-empty texts, however, cannot be derived to* ε *(#3). Star-operators are also derivable to* ε, *so empty text nodes are derivable to arbitrary star-operators (#4).*

(#2) $\mathcal{E}[\![$ `TxtR "", Epsilon` $]\!]\mu :=$ `True`
(#3) $\mathcal{E}[\![$ `TxtR _, Epsilon` $]\!]\mu :=$ `False`
(#4) $\mathcal{E}[\![$ `TxtR "", Star _` $]\!]\mu :=$ `True`

It still is possible (#6), that ‘xtl:text’ *in the instance can generate arbitrary text as output.*

(#6) $\mathcal{E}[\![$ `TxtR text, TextR _` $]\!]\mu :=$ `True`

Comparing two text nodes (#7) is trivial, the same as validation of a text node against an arbitrary element node (#8).

(#7) $\mathcal{E}[\![$ `TxtR text1, TxtR text2` $]\!]\mu :=$
`text1 == text2`
(#8) $\mathcal{E}[\![$ `TxtR text, ElR _ _ _` $]\!]\mu :=$ `False`

The validation of a string against a hedge is similar to the validation of element nodes against a hedge (#5). A repeating pattern is searched. If no non-deterministically obtained partitions matches, so the considered string may only derive to ε.

(#5) $\mathcal{E}[\![$ `TxtR text, Star r` $]\!]\mu :=$
`if (h==True) then True else`
$\mathcal{E}[\![$ `TxtR text, Epsilon` $]\!]\mu$
for `h ==` $\vee$ `[True|`
`(s1,s2)` $\leftarrow \overline{\textit{frontSplitText}}^{1}$ `text`
$\wedge\ \mathcal{E}[\![$ `TxtR s1, r` $]\!]\mu$
$\wedge\ \mathcal{E}[\![$ `TxtR s2, Star r` $]\!]$ `]`

The validation against a hedge is only valid if a sound, possibly empty partition of a hedge exists, so both texts concatenated equals the wanted text.

(#1) $\mathcal{E}[\![$ `TxtR text, Then r1 r2` $]\!]\mu :=$
$\vee$ `[True|`
`(s1,s2)` $\leftarrow \overline{\textit{splitText}}^{1}$ `text`
$\wedge\ \mathcal{E}[\![$ `TxtR s1, r1` $]\!]\mu$
$\wedge\ \mathcal{E}[\![$ `TxtR s2, r2` $]\!]\mu$ `]`

The fourth passage considers element nodes as an instance. Element nodes are in contrast to text nodes atomic. It means an element cannot be part of another element node at the same level. This atomicity leads in (ElR8), (ElR9) and (ElR10) to schema-tags are excluded from the very beginning.

(ElR8) $\mathcal{E}[\![$ `ElR _ _ _, TextR _` $]\!]\mu :=$ `False`
(ElR9) $\mathcal{E}[\![$ `ElR _ _ _, Epsilon` $]\!]\mu :=$ `False`
(ElR10) $\mathcal{E}[\![$ `ElR _ _ _, TxtR _` $]\!]\mu :=$ `False`

Case (ElR7) states an element node validates against a star-operator only if the subexpression underneath validates against the instance node. So, the loop body's beginning and end nodes are derivable to ε, *and there is only the element node between them.*

(ElR7) $\mathcal{E}[\![$ `ElR name atts r, Star s` $]\!]\mu :=$
$\mathcal{E}[\![$ `ElR name atts r, s` $]\!]$

Rule (ElR1) may only look trivial at first glance. It validates an element node against another. However, it is crucial to notice that besides equality of the element name, there is also the set of existing schema nodes that need to match after the canonisation. In the case of schema nodes, ‘AttrR’ *also need to be considered. In any case, validation has to continue with the reduced schema hedge* `r3`.

(ElR1) $\mathcal{E}[\![$ `L, R` $]\!]\mu :=$
`L=ElR name1 atts1 r1,`
`R=ElR name2 atts2 r2,`
`(name1 ==` $\overline{\textit{qSort}}^{1}$ `atts1 == atts3)`
$\wedge\ \mathcal{E}[\![$ `r1, r3` $]\!]\mu$
for `(ElR _ atts3 r3) =`
$\overline{\textit{extractAttributes}}^{1}$ `(ElR name2 atts2 r2)`

Validation against `Then` *cannot just proceed. It needs to answer first the question of which element ought to be validated first. So,* $\pi(\mathbb{S})$ *would need to be determined, which is not, at least not initially (see sect.III). So the left branch of* ‘Then’ *needs to be checked. Depending on that, the cases (ElR2), (ElR3), (ElR4) and (ElR6) result for all remaining schema nodes.*

(ElR6) $\mathcal{E}[\![$ `ElR _ _ _, Then _ _` $]\!]\mu :=$ `False`

The cases (ElR2), (ElR3) and (ElR4) are obvious and do not require further explanations.

(ElR2) $\mathcal{E}[\![$ `L, R` $]\!]\mu :=$
`L=ElR name1 atts1 r1,`
`R=Then (ElR name2 atts2 r2) s,`
$\mathcal{E}[\![$ `L, ElR name2 atts2 r2` $]\!]\mu$
$\wedge\ \mathcal{E}[\![$ `Epsilon, s` $]\!]\mu$

(ElR3) $\mathcal{E}[\![$ `L, R` $]\!]\mu :=$

L=ElR $name1$ $atts1$ $r1$,
R=Then (Or $s1$ $s2$) s,
$(\mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, Or $s1$ $s2$$]\!]\mu$
$\wedge\, \mathcal{E}[\![$Epsilon, $s$$]\!]\mu)$
$\vee\, (\mathcal{E}[\![$Epsilon, Or $s1$ $s2$$]\!]\mu$
$\wedge\, \mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, $s$$]\!]\mu)$

(ElR4) $\mathcal{E}[\![L, R]\!]\mu :=$
L=ElR $name1$ $atts1$ $r1$,
R=Then (Star $s1$) s,
$(\mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, $s1$$]\!]\mu$
$\wedge\, \mathcal{E}[\![$Epsilon, $s$$]\!]\mu)$
$\vee \mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, $s$$]\!]\mu$

In the case of (ElR5), it is checked if the element node from the instance is in the macro body. – If so, the following hedge s has to be derivable to ε. Alternatively, the element node is in s. Then the macro body must be derivable to ε.

(ElR5) $\mathcal{E}[\![L, R]\!]\mu :=$
L=ElR $name1$ $atts1$ $r1$,
R=Then (MacroR m) s,
$(\mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, MacroR $m$$]\!]\mu$
$\wedge\, \mathcal{E}[\![$Epsilon, $s$$]\!]\mu)$
$\vee\, (\mathcal{E}[\![$Epsilon, MacroR $m$$]\!]\mu$
$\wedge\, \mathcal{E}[\![$ElR $name1$ $atts1$ $r1$, $s$$]\!]\mu)$

Both rules (Φ) and (Ω) are universal w.r.t. instance nodes. A macro call in the schema has – independent from the actual instance node – an unfolding or substitution by the macro body in consequence, which is represented by exactly one OBDD-expression.

(Φ) $\mathcal{E}[\![$$inst$, MacroR $mname$$]\!]\mu :=$
$\mathcal{E}[\![$$inst$,$word$$]\!]\mu$
for $word$ = $\overline{\textit{getMacro}}^{2}$ $mname$ μ

The same is with 'Or' in the schema – independent from the concrete instance a matching case $r1$ or case $r2$ is validated.

(Ω) $\mathcal{E}[\![$$inst$, Or $r1$ $r2$$]\!]\mu :=$
$\mathcal{E}[\![$$inst$, $r1$$]\!]\mu \vee \mathcal{E}[\![$$inst$, $r2$$]\!]\mu$

Example

The following example shows the validation of a simple document. Let the schema s be given from fig.8(a) and a corresponding instance document i from fig.8(b).

The textual notations are:

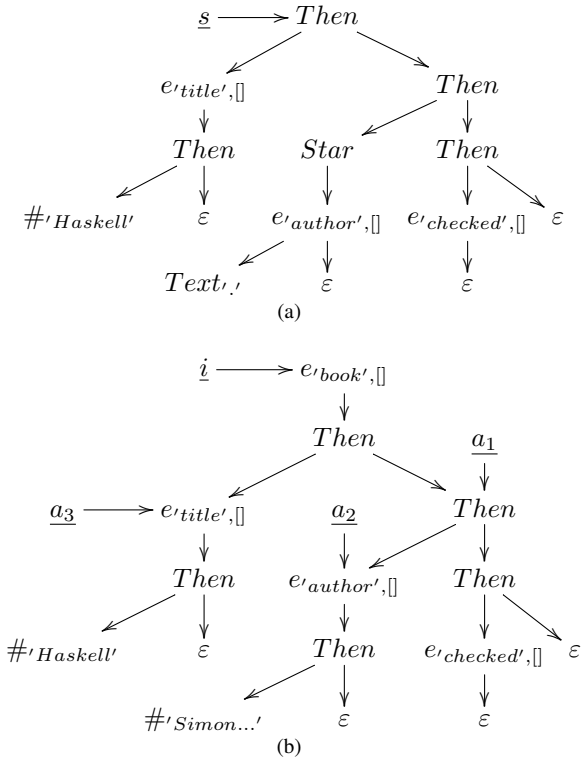


Fig. 8. Examples for Regular Expressions

```
s  =  ElR "book"  [] Then (ElR
      "title"  Then (TxtR "Haskell")
      Epsilon) Then (Star (ElR
      "author"  [] (Then (TextR ".")
      Epsilon))) Then (ElR "checked"
      [] Epsilon) Epsilon

i  =  ElR "book"  [] Then (ElR
      "title"  Then (TxtR "Haskell")
      Epsilon) Then (ElR "author"
      [] Then (TxtR "Simon...")
      Epsilon) Then (ElR "checked"
      [] Epsilon) Epsilon

a1 =  Then (ElR "author"  [] Then
      (TxtR "Simon...") Epsilon)
      Epsilon

a2 =  ElR "author"  [] Then (TxtR
      "Simon...") Epsilon

a3 =  ElR "title"  [] Then (TxtR
      "Haskell") Epsilon
```

The derivation looks as following:

$\mathcal{E}[\![i, s]\!]$
= $\mathcal{E}[\![L,$ `ElR "book" [] ...`$]\!]$
L=`ElR "book" [] ...`
= (`"book"=="book"`) $\wedge$ $\overline{qSort}^1$ `[]==atts3`)
(ELR1) $\wedge$ $\mathcal{E}[\![$`Then (ElR "title" [] ..., r3)`$]\!]$
for (`ElR _ atts3 r3`) = $\overline{extractAttributes}^1$ `s`
= (`[]==atts3`)
$\wedge\mathcal{E}[\![$`Then (ElR "title" [] ..., r3`$]\!]$
for (`ElR _ atts3 r3`) = `s`
= $\mathcal{E}[\![$`Then (ElR "title" ..., Then (R`$]\!]$
R=`ElR "title"[] ...`
= $\vee$`[ True|`
(Then7) (`t1, t2`) $\leftarrow$ $\overline{splits}^1$(`Then L R`)
L=`(ElR "title" ...),`
R=`Then (ElR "author" ...),`
$\wedge$ $\mathcal{E}[\![$`t1, ElR "title" ...`$]\!]$
$\wedge$ $\mathcal{E}[\![$`t2, Then (Star ...`$]\!]$ `]`
= $\vee$`[ True|` $\mathcal{E}[\![L, R]\!]$,
(nd) L=`ElR "title" [] ...,`
R=`ElR "title" ...,`
$\wedge$ $\mathcal{E}[\![L2, R2]\!]$,
$L2$=`Then (ElR "author" [] ..,`
$R2$=`Then (Star ... ]`
= $\mathcal{E}[\![L, R]\!]$
L= `ElR "title" [] ...,`
R= `ElR "title" ...,`
$\wedge\mathcal{E}[\![L2,$ `Then (Star ...`$]\!]$,
$L2$= `Then (ElR "author" [] ...`
= ...
= $\mathcal{E}[\![$`Then (ElR "author" [] ..., R`$]\!]$
R= `Then (Star ...`
= $\vee$`[True|`
(Then7) (`t1, t2`) $\leftarrow$ $\overline{splits}^1$(`Then L Then ...`),
L=`(ElR "author" ...),`
$\wedge$ $\mathcal{E}[\![$`t1, Star (ElR ...)`$]\!]$
$\wedge$ $\mathcal{E}[\![$`t2, Then R Epsilon`$]\!]$
R= `(ElR "checked" [] Epsilon) ]`
= $\vee$`[ True|` $\mathcal{E}[\![$`a1, S`$]\!]$,
S=`Star (ElR "author" [] R),`
R=`Then (TextR ".") Epsilon,`
$\wedge$ $\mathcal{E}[\![L2, R2]\!]$,
$L2$= `Then (ElR ...) Epsilon,`
$R2$= `Then (ElR ...) Epsilon ]`
= $\mathcal{E}[\![$`a1, Star (ElR "author" [] R)`$]\!]$
R= `Then (TextR ".") Epsilon`
$\wedge\mathcal{E}[\![$`Then (ElR ...) Epsilon, R2`$]\!]$
$R2$=`Then (ElR ...) Epsilon`
= $\mathcal{E}[\![$`a1, Star (ElR "author" [] R)`$]\!]$
(Then6) R= `Then (TextR ".") Epsilon`
$\wedge\mathcal{E}[\![$`ElR "checked" [] Epsilon, R2`$]\!]$
$R2$=`Then (ElR "checked" []`
`Epsilon) Epsilon`
= ...

= $\mathcal{E}[\![$`a1, Star (ElR "author" [] R)`$]\!]$
R=`Then (TextR ".") Epsilon`
= $\mathcal{E}[\![$`a2, Star (ElR "author" [] R)`$]\!]$
(Then6) R=`Then (TextR ".") Epsilon`
= $\mathcal{E}[\![$`a2, ElR "author" [] R`$]\!]$
(Then7) R=`Then (TextR ".") Epsilon`
= (`"author" =="author"`)
(ElR1) $\wedge$ (`qSort`^{1} `[]==atts3`)
$\wedge$ $\mathcal{E}[\![$`Then (TxtR "Simon...") Eps, r3`$]\!]$
for (`ElR _ atts3 r3`) =
$\overline{extractAttributes}^1$(`ElR "author" [] Then`
`(TextR ".") Epsilon`)
= (`qSort`^{1} `[]==atts3`)
$\wedge$ $\mathcal{E}[\![$`Then (TxtR "Simon...") Eps, r3`$]\!]$
for (`ElR _ atts3 r3`) = `ElR "author"`
`[] Then (TextR ".") Epsilon`
= $\mathcal{E}[\![$`Then (TxtR "Simon...") Eps, R`$]\!]$
R=`Then (TextR ".") Eps`
= $\mathcal{E}[\![$`TxtR "Simon...", R`$]\!]$
(Then6) R=`Then (TextR ".") Eps`

= $\vee$ `[True|`
(#1) (`s1, s2`) $\leftarrow$ $\overline{splitText}^1$ `"Simon..."`
$\wedge$ $\mathcal{E}[\![$`TxtR s1, TextR "."`$]\!]$
$\wedge$ $\mathcal{E}[\![$`TxtR s2, Epsilon`$]\!]$`]`
= $\mathcal{E}[\![$`TxtR "Simon...", TextR "."`$]\!]$
$\wedge$ $\mathcal{E}[\![$`TxtR "", Epsilon`$]\!]$
= $\mathcal{E}[\![$`TxtR "", Epsilon`$]\!]$
(#6)
= `True` ■
(#2)

$\mathcal{E}[\![$`a3, a3`$]\!]$
= $\mathcal{E}[\![$`Then (TxtR "Haskell") Eps, r3`$]\!]$
(ElR1) *for* (`ElR _ atts3 r3`) =
$\overline{extractAttributes}^1$(`ElR "title" [] Then`
`(TxtR "Haskell") Epsilon`)
= (`qSort`^{1} `[]==atts3`)
$\wedge$ $\mathcal{E}[\![$`Then (TxtR "Haskell") Eps, r3`$]\!]$
for (`ElR _ atts3 r3`) =
`ElR "title" [] R`
`R=Then (TxtR "Haskell") Epsilon`
= $\mathcal{E}[\![$`L, Then (TxtR "Haskell") Eps`$]\!]$
`L=Then (TxtR "Haskell") Eps`
= $\mathcal{E}[\![$`TxtR "Haskell", R`$]\!]$
(Then6) `R=Then (TxtR "Haskell") Epsilon`
= $\vee$ `[True|`(`s1, s2`) $\leftarrow$ $\overline{splitText}^1$ `"Haskell.."`
(#1) $\wedge\mathcal{E}[\![$`TxtR s1, TxtR "Haskell"`$]\!]$
$\wedge\mathcal{E}[\![$`TxtR s2, Epsilon`$]\!]$ `]`
= $\vee$ `[True|`
$\mathcal{E}[\![$`TxtR "Haskell", TxtR "Haskell"`$]\!]$
$\wedge\mathcal{E}[\![$`TxtR "", Epsilon`$]\!]$`]`
= $\mathcal{E}[\![$`TxtR "Haskell", TxtR "Haskell"`$]\!]$
$\wedge\mathcal{E}[\![$`TxtR "", Epsilon`$]\!]$
= (`"Haskell"=="Haskell"`)
(#7) $\wedge\mathcal{E}[\![$`TxtR "", Epsilon`$]\!]$
= `True` ■
(#2)

$\mathcal{E}[\![$`ElR "checked" [] Eps, R`$]\!]$
R=`Then (ElR "checked" [] Eps) Eps`
= $\mathcal{E}[\![L,\ L]\!]$
(ElR2) L=`ElR "checked" [] Epsilon`
$\wedge\mathcal{E}[\![$`Epsilon, Epsilon`$]\!]$
= (`"checked"=="checked"`)
(ElR1) $\wedge$(*qSort*$'$ `[]==atts3`)
$\wedge\mathcal{E}[\![$`Epsilon, r3`$]\!]$
for `(ElR _ atts3 r3)` =
extractAttributes$'$ `(ElR "checked" [] Eps)`
= (*qSort*$'$ `[]==atts3`) $\wedge\ \mathcal{E}[\![$`Epsilon, r3`$]\!]$
for `(ElR _ atts3 r3)` =
`ElR "checked" [] Epsilon`
= `True` ■

In the derivation below the equality sign the applied validation rules are provided. Rules labelled with '`(nd)`*' indicate non-deterministic selection is made. When searching for a solution, the program requires numerous executions, which have to be refined each time. The rule '*`(nd)`*', therefore, is only of didactic help. The continuations '. . .' are shortened regular expressions — those complete preceding rules.*

Although the selected instance document is relatively small, the derivation shows that even a few non-deterministic cases can lead to extensive search.

V. Implementation

A. Overview

Haskell is recommended for implementing denotational semantics because of the lack of side-effects and its functional paradigm. Haskell's features are very close to the syntax and semantic denotational semantics. It includes, for instance, higher-order functions, strict static typing, data encapsulation, lazy evaluation and generic polymorphism [65], [62]. However, some functional programming languages, like LISP or Miranda do not have at least one of the mentioned advantages. That is why Haskell is chosen.

The Haskell XML-Toolbox (HXT) [58] provides a huge library of functions processing XML. In version 7.0 HXT contains over 114 non-empty sub-packages. It includes an XML data model, XML parser and serialiser, a validator for RelaxNG and DTD schemas and processors for XPath and XSLT. Apart from that, HXT has numerous navigation function and constructors. The function '`readDocument`*' does parsing. Serialisation is done by '*`writeDocument`*'. Both functions use stateful arrows (cf. sect.III), which guarantees referential transparency to the outside. Both functions obey a strict sequential evaluation ordering. Filters may be used as a substitution for arrows. Filters can be used without the binary sequential operator '>>>' (comparable to "';"' in C or Pascal). It allows lazy evaluation, and not needed calculations may be dropped. HXT is a toolbox. HXT does not come as a framework because control always sticks to the application programmer and never changes. Just a few filters allow semi-automatic processing of XML documents by using user-defined helper functions.*

Localisation of files works with URIs-addressing. Thus XML sources are independently addressable from the underlying system. Unfortunately, the recent HTTP module does not support relative addressing to the full extend.

In contrast, HaXML [68] includes 25 non-empty sub-packages. HaXML is reduced to essential XML-operations, a pretty-printer and HTML-processing. It uses filters as combinators. Arrows are currently not foreseen. Despite that, the integrated HaXML-parser is based on memoisation. So, needed previously calculated subexpressions are calculated only once.

The selection for the right toolkit wins HXT due to its huge amount of supported features.

B. Architecture

This section introduces the main functions and modules written in Haskell. The implementation of instantiator and validator are described briefly. Apart from that, tests are shortly demonstrated in order to assure proper implementations. Introduced models are checked visually and briefly.

1) Function Dependency Graph: The function dependency graph for each instantiator and validator is illustrated in fig.9. Essentially, the implementation consists of three modules: `Main`*,* `Instantiator` *and* `Validator`*. Helper functions as '*`getXPathSubTrees`*' are skipped.*

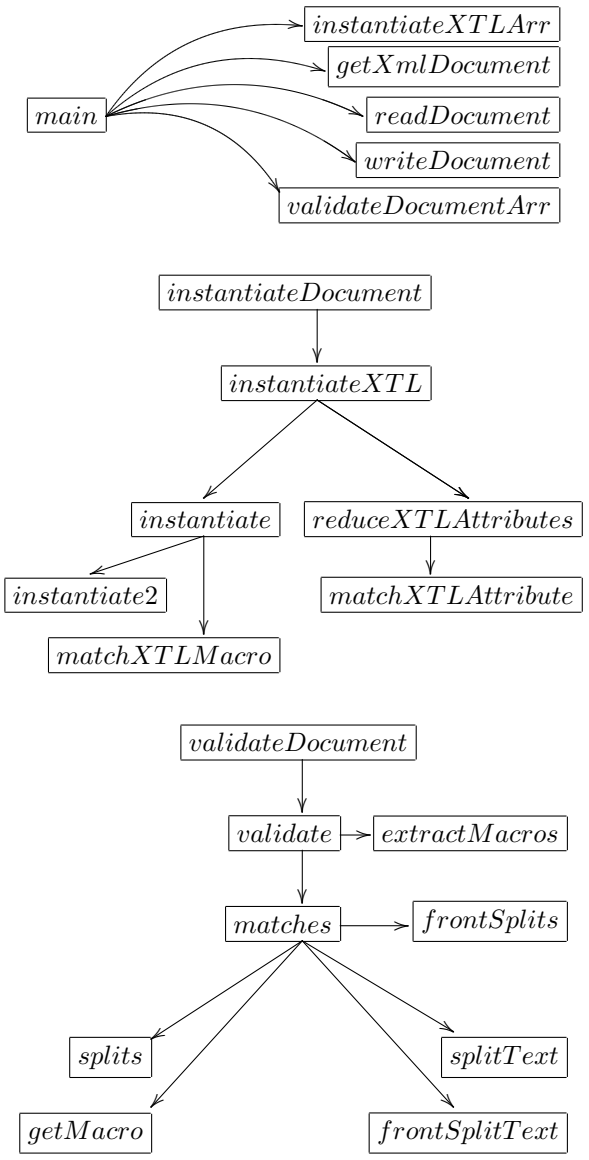


Fig. 9. Function Dependency Graph for `Instantiator` and `Validator`

Functions for instantiation and validation are located in module `Main`. Here the functions that appear filled are used by both programs. In contrast to function '`readDocument`', '`getXmlDocument`' reads XML-documents by avoiding arrows, so further XML-documents in '`main`' can be processed.

The function '`instantiateXTL`' performs an instantiation using a template and a source for instantiation data. '`validateDocument`' performs validation of an instance document against an XTL-schema. Both functions are implemented as arrows. These are in '`main`' together with input and output function in a '`do`'-environment. Because '`instantiateDocument`' and '`validateDocument`' require a second document, the effective function is implemented as partially defined arrows.

'`instantiateDocument`' *transforms multiple '`XmlTree`' into the data model '`XTL`'.*

It corresponds to $\mathcal{E}^{Start}[\![]\!]$ in terms of denotational semantics. An instantiation is triggered by '`instantiateXTL`'. A pre-calculation step within the top-level node of the template XTL-attributes is united with preexisting attributes from element nodes. That corresponds to $\mathcal{E}^{\tau}[\![]\!]$. Then instantiation resumes recursively. For an element node, this is described by the function '`instantiate`' (namely $\mathcal{E}[\![]\!]$) and '`instantiate2`' ($\mathcal{E}^{\alpha}[\![]\!]$) describes it for hedges. '`matchXTLMacro`' distinguishes on the top-level for an element node, if it is a macro or not.

Validation is triggered by '`validate`' as soon as a given '`XTL`' is transferred to the regular data model '`Reg`'. Validation equals $\mathcal{E}^{Start}[\![]\!]$ in terms of denotational semantics. A call to '`extractMacros`' filters at the beginning of all macros. It corresponds to $\mathcal{E}^{\tau}[\![]\!]$. Validation continues recursively with '`matches`' with regular expressions for both instance and schema. The function '`matches`' corresponds to $\mathcal{E}[\![]\!]$. Macro calls are implemented by '`getMacro`'. The non-deterministic splitting of interleavings is done by '`frontSplitText`', '`splitText`', '`splits`' and '`frontSplits`' (cf. sect.III-A3) and depends on the node type for each element node.

Both implementations of both processes strictly follow the denotational semantics from the appendix. So, on validation, match-rule (E1) is noted in Haskell as following (see sect.IV):

```
matches Epsilon (TxtR "") macros = True.
```

The same holds for instantiation (cf. app.IX).

Not well-formed documents cannot be instantiated. Since '`readDocument`' parses lazily and traverses XML documents in pre-order, errors are issued with a position. Element nodes, which do not obey conventions from sect.II-C, are interpreted as usual element nodes (cf. [32]).

As presented in sect.IV-B2, instantiation data shall be passed in a context in '`xtl:for-each`'. Binding instantiation data implicitly to the PHP function is disadvantageous — children of '`xtl:for-each`' access portions of instantiation data.

Errors are issued locally. However, this does not mean a thrown error is the reason for a failure during the validation (cf. sect.IV). The pre-order processed schema is serialised pre-order to meet the requirement of error localisation — this allows a faster localisation by the user. An error stack would simplify bug tracking on the one side. On the other side, it would, however, rapidly increase execution time. The serialised document contains the last valid node because validation operates lazy.

2) Checking Validity: The proof for a correct model transformation was done in sect.IV-A. The proof for completeness of the data models is already done in sect.IV. Here, all XML-document composed of text and element nodes can be expressed by '`XTL`' and '`Reg`'. Properties of instantiation and validation are discussed in sect.III.

Correctness of the implementations is guaranteed by a precise translation of the denotational semantic and Haskell's referential transparency (cf. sect.IV-B2, IV-C1). Implementation validity is assured by HUnit-tests [35]. During instantiation, XPath is prototypically used as a command language. Tests cover the rules of the denotational semantics and also in combination with other rules. Tests check for valid and invalid input. The validator has currently 818 tests, where the instantiator has 62 tests. The big discrepancy in the exponential rise of complexity is due to the additional transitions by the validator. The instantiator is determined and much simpler in comparison to the validator.

Documentation for the Haskell functions is given by a Haddock-helpfile [47].

As suggested in sect.IV, test cases should be enriched by infinite documents. Infinite OBDDs can be passed to `'validate'`*-functions. The termination behaviour may practically be restricted, since in case of a left-recursion, it is for sure after a finite number of steps a result eventually is available, or a non-termination '⊥' is the "'result"'. The following example shows infinite data structures as an extension to existing test cases:*

```
genTree,genTree2,genTree3::Tree String
genTree = Node "a" (Node "b" genTree Eps)
                   (Node "c" genTree Eps)
genTree2 = Node "a" (Node "b" Eps genTree2)
                    (Node "c" genTree2 Eps)
genTree3 = Node "a" Eps
            (Node "b" genTree3 Eps)

test Eps _ = True
test (Node _ l r) c =
    if (c==10) True
    else or [(test l (c+1)), (test r (c+1))]
```

For the sake of a clear explanation, a binary tree `'Tree'` *is used here. This tree has a similar structure as regular expressions* `'Then'` *and* `'Or'` *in* `'Reg'` *(cf. sect.IV-A3):*

The underlying data model `Tree` *is defined as*

```
data Tree a = Node a (Tree a) (Tree a) | Eps.
```

The test function `'test'` *has the type* `'Tree a→Integer→Bool'`*. The starting value* `c==0` *iterates a polymorphic infinite* `'Tree'` *until a node of depth ten is visited for the first time. The equivalence partition of positive test cases encloses all (*`Tree Integer, c`*), where 'c' is an integer less equals 10. For this partition, the function terminates with the result* `'True'`*. For* `c>10`*, it results in '⊥', because the data structure is infinite and the base case "'*`test Eps _= True`*"' is unreachable. This example demonstrates the meaning of infinite OBDDs as a test case. If the test function were passed a negative finite input, it would always return a result not equal to '⊥'. For an invalid input, the test function returns '⊥'. Match-rules of the validator returning a result for '⊥', and '⊥' otherwise have similar behaviour. That is why test functions should consider selected unlimited test data or data structures as* `'genTree'`*,* `'genTree2'` *and* `'genTree3'`*.*

C. Optimisations

Recommendations for improvement on two examples in Haskell are shown.

1) Multiple Iterations: Whilst instantiation hedges are iterated several times in order to filter nodes with a particular property. The rules (E) and (I3) are considered patterns from app.IX:

$$\textbf{let}\ \underline{attDefs} = \overline{filter}^{2}(\lambda \underline{child}.\mathcal{E}^{MA}[\![\underline{child}]\!])\underline{c},$$
$$\underline{nodes} = \overline{filter}^{2}(\lambda \underline{child}.\overline{not}^{1}\ \mathcal{E}^{MA}[\![\underline{child}]\!])\underline{c}\ .$$

It would be more efficient to comprehend both resulting variables, which would be a tuple, and accumulate one of both tuple-variants. The following fragment shows a corresponding improvement towards reducing the number of iterations:

$$\textbf{let}\ (\underline{attDefs},\ \underline{nodes}) =$$
$$\overline{foldl}^{3}(\lambda(\underline{m},\underline{n})\underline{c}.\textbf{if}\ \mathcal{E}^{MA}[\![\underline{c}]\!]\ \textbf{then}\ (\underline{m}\mathbin{+\!\!+}[\underline{c}],\underline{n})$$
$$\textbf{else}\ (\underline{m},\underline{n}\mathbin{+\!\!+}[\underline{c}]))\ \texttt{([],[])}\ \underline{c}\ .$$

By bundling, child nodes are processed once. Lazy evaluation still holds, but in every case, `'let'` *is evaluated before the function body. The performance bargain looks reasonable initially because the new variant has an alternative with a condition and alternative. A rather complex program buys the supposed advantage in performance since the new variant has an alternative condition. The rows, which initially used to be relatively short, still hardly differ. The gained improvement has an unsymmetrical behaviour.*

The separation between program and optimisation rules can be achieved in GHC by introducing an additional Haskell program comment. This comment is checked while interpretation and matching optimisation rules are applied to defined functions.

Concerning sect.V-D, this simplification has little meaning because object-oriented modelling interprets operations over aggregated child nodes undoubtedly different.

2) Lazy Evaluation: Left-associative folds have a performance advantage over right-associative folds because of the lazy evaluation. Because of the homomorphism-condition holding during validation (see sect.III-A4), regular subexpression may be evaluated in arbitrary ordering. Subexpressions, however, may be skipped. So, list comprehensions, which are left-associative, lead to `'frontSplits'` *calculates all partitions are ascending by the length of a regular expression. This calculation skips partitions quicker.*

Hash-tables should be used to access attribute entries during validation because attributes are accessed quite often and those often not canonised in applications. The algorithm from app.IX uses a lazy evaluation strategy, speeding up further using hash-tables. The table size should grow in proportion to the derived regular expression length (see sect.IV).

D. Implementation in Java

Before the Haskell implementation is translated into Java, several main caveats need to be considered. Xerces [70] may be used for the input and output of XML documents. JXPath [40] may be used for XPath-queries.

Data models 'XTL' and 'Reg', which are given as algebraic data types, must first be modelled as parametrised classes. So, for 'Reg', an abstract class 'RegEx' is defined. 'RegEx' has subclasses, for instance, 'Then' and 'Or'.

Haskell's generic polymorphism is restricted further by ad-hoc polymorphism [22] and is used, for instance, by sorting and when processing instantiation data. Java 5.0 provides the possibility to replace generic polymorphism with templates that are determined on runtime.

Unfortunately, lazy evaluation cannot directly be simulated in Java. By explicit checks, the evaluation order needs to be influenced, s.t. many cases are eliminated. Alternatively, the most likely cases need to be checked. For the encapsulation of lazy methods, the STRATEGY pattern is promising.

Higher-order function needs to be mimicked by polymorph classes. Here, HOFs are implemented as concrete classes, which calls polymorphic class methods. Partial functions can be mimicked in Java without sending messages. Partial arguments can be determined by queries to 'get'-methods on the called object. Interfaces can describe static typing of the PHP function.

Matching rules of the validation algorithm can either be adopted as is. In this case, the validator operates recursively descendant and consumes many resources. The alternative is to consider a pushdown-automaton. However, the overall functional paradigm will severely change, and so will the denotational semantics. In this case, it may be better to refer to operational semantics instead. However, this is not the aim of this work (see sect.III-A).

1) Proposition of Class Diagram: Fig.V-D1 shows a possible design for validation. The class `Validator` represents the caller, which initiates an instance document against a given schema document. The class `RegEx` is abstract, represents regular expressions, and currently has eight subclasses. Subclasses containing one or two `RegEx` allow variable child nodes. The abstract methods `validate` and `getIdentifier` are implemented in the subclasses. The method `getIdentifier` returns the identifier of a subclass. This method breaks encapsulation. However, it may be tolerated if all regular expressions are fully covered. By object-centric identifiers, every `validate` method can be implemented by switch-constructs. Implementation follows the rules of the corresponding denotational semantics closely herewith.

Since the methods `validateOr` and `validateMacro` are universal (see sect.IV-C1), `RegEx` can implement those. The class `MacroEntry` represents an entry of the macro-environment μ (see sect.II-C).

In this class diagram, several design and architectural patterns are hidden (after Fowler [27] and Kernievsky [42]).

Initially, regular expressions are constructed by the counted subclasses by using the BUILDER pattern. Here `Validator` acts as Director and `RegEx` as AbstractBuilder. The standard method is `create`. The DECORATOR pattern allows some `RegEx`-classes the reuse of previously implemented code. The classes `TxtR`, `Epsilon` and `TextR` act as ConcreteComponent, `RegEx` as Component and `ElR`, `Star`, `MacroR`, `Or`, `Then` as ConcreteDecorator.

The INTERPRETER pattern describes a regular language. The client `Validator` triggers validation by calling the method `validate`. The IMPLICIT-LANGUAGE pattern consists of terminals and non-terminals, which generate the language. Those subclasses that do not contain composition act as Terminals Classes, which have a simple composition `RegEx`, act as Non-Terminal. Both symbols are interpretable expressions.

On validation, each node type is matched with an instance node. Invalid combinators are skipped and validated against 'False'. The classes do not implement `validate` for every `RegEx`-subclass. That is also because the structure of regular expression for schemas does not extend in a meaningful way. Hence, each class type during evaluation returns identifiers or type information. The function `getIdentifier` can obtain this information. That is why a separation between the data model and tree traversal does not make sense.

Hence, an implementation by the VISITOR pattern is not appropriate here. Besides, the universal DESCRIPTOR pattern occurs whose INSTANCE is the class `TextSplitting` and whose DESCRIPTION is `TxtR`. In analogy, `ElR` and `AttrEntry` also match the DESCRIPTOR pattern.

Apart from this, the TEMPLATE-METHOD pattern can be found. The abstract class here is `RegEx`, where `validateMacro` and `validateOr` act as Template. The subclasses listing the same methods are Hooks, which are referring to templates.

In the next step, the data model 'XTL' and polymorphic instantiation data and the instantiator can be modelled. The proposed test suite can be used in addition to the module test.

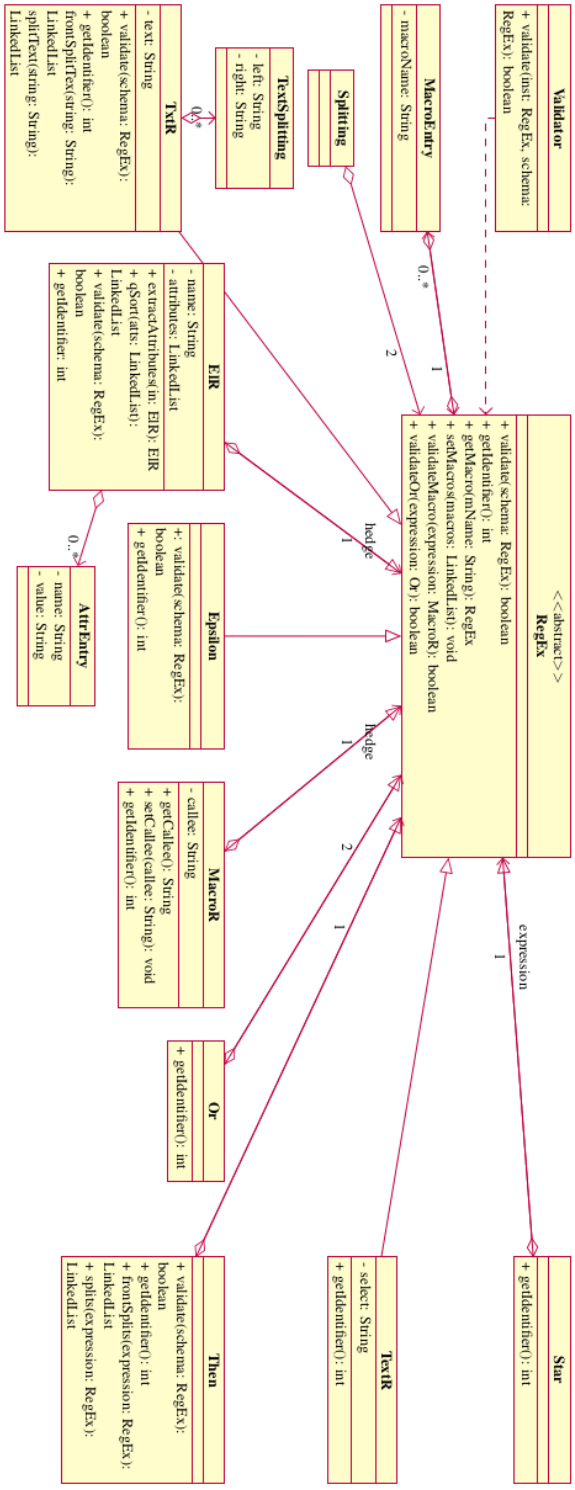


Fig. 10. Validator Design for Java

VI. Comparison

In this section, dedicated schema languages are compared according to previously selected criteria. Here, language features are more of importance than implementations. The following questions will be taken into consideration:

1) *What are the positive factors towards unification?*
   Here, semantics, syntax and algorithms for both instantiation and validation shall be considered. The criteria weights shall be considered thoroughly. Criteria in favour and against unification shall be clarified.
2) *Which linguistic features are in favour and which ones are against unification?*
   Is it possible to adapt and adequately represent features from other schema languages? A metric shall be provided if possible. If the other XML schema languages show up weakness, investigate if the weakness may be resolved.
3) *Which consequences do unification have in practice?*
   What is the practical benefit of unification?

Furthermore, differences and mutual grounds of features towards a unification from previous sections shall thoroughly be analysed.

A. Considered Languages

The considered languages are illustrated in fig.11. It is XTL [32] which may be considered as one fraction, and rather popular schema languages like XSD [28], [60], [7], RelaxNG [20] and DTD [41]. Since DTD may not be described within XML, DTD will only be referred to at some positions only where appropriate for comparison purposes.

XSLT is considered for the sake of comparison between template and schema languages [19].

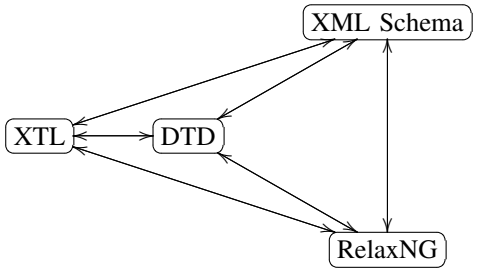


Fig. 11. Considered Schema Languages

B. Criteria

The criteria are applied to XTL as well as to selected schema languages. Criteria are supposed to be as common as possible and are focused on unification. In conclusion, comparisons are mostly qualitatively.

A unification of template expansion and schema validation is achieved by involved documents and referred schema languages (cf. sect.III). Orthogonality and distributivity are properties that have little meaning here. Both properties may already be covered by modularity.

Similarity

Similarity refers to template and instance documents on the one side and template and schema documents on the other side. If the template and schema document are the same, then unification is achieved. Otherwise, there are commands in the template language which do not correspond to commands in the schema language or vice versa. That is distinguishing instantiation from validation.

Moreover, similarity means an adequate representation of a template document w.r.t. an instance document. If both are too different, then the similarity is too little. It means template markups can be transferred into instance markups by only a considerable amount of changes.

If template and schema document or template and instance document were only close enough, then a general unification is achievable.

Expressibility

Expressibility asks if features may express a template language from the schema language and vice versa.

Regularity properties and type safety are researched for unification. An essential question to find here is whether functions and symbols can be validated as expression.

Even the representation of rules themselves may influence the unification. For instance, one question emerges if rule representation may improve unification if rules are based purely on filters and pattern-matching. It was observed that pattern-matching often is better for more compact notation, if possible at all, than filters, for instance.

1) Classification: The taxonomies are shown in tab.X, XI may be derived from sect.I. The three views are mostly related to schema languages.

Syntax	Semantic	Complexity
Schema-Style Ordering Syntactic Sugar Pattern-Matching Regularity Symbols Control Flow	Typing Functions Constraints Error Handling	Time- / Memory usage Automata Class Evaluation Ordering

TABLE X
TAXONOMY FROM THE PERSPECTIVE OF A SCHEMA LANGUAGE

Tab.X differentiates between schema features' syntax, semantic validation concepts, and corresponding algorithms' footprints. The column "'Syntax"' denotes "'Symbols"' hedges and attribute unions. Since hedges and attributes of templates and schemas do not alter, the corresponding identifiers may be considered symbols. The remaining columns are not ambiguous (see sect.II). The existing column "'Typing"' ask whether generated elements always have a type and if validation may result in an intersection of command-tags. "'Functions"' expresses representation and interpretation of functions which have an arity of one or more. Constraints are a possibility to restrict nodes and texts even more. Granularity plays a key role here. "'Syntactic Sugar"' means if, for instance, parametrised loops and other recurring idioms may be replaced by a more elegant syntactic notation.

Automata class is determined by the level of determinism and evaluation order. The class is a measure of how complex a concrete validation implementation may be.

The taxonomy (a) from tab.XI splits the comparison criteria from (an application) developer's view. The level of extensibility and variability determines the partition. Modules are on possibility in order to bind arbitrary schemas by namespaces and symbols. Variability means, for example, how much a command language, instantiation data and the evaluation order might be influenced. "'Evaluation Order"' means if expressions are evaluated one after another in the order they appear in the incoming document or if they are only used when needed.

Taxonomy (b) compares subjective and objective criteria. Self-explanatory means similarity to a document. However, it may also mean a concise description of a schema node.

2) Assessment: In this section, comparisons are based on tab.X. Taxonomies (a) and (b) of tab.XI are compared with the first table. The criteria from tab.X are loosely coupled. Constraints and typing are of overall practical meaning. "'Error handling"' is a secondary requirement to usability. Functions are of utmost importance to the expressibility of template languages.

Syntax summarises the most important criteria in comparison to semantic and complexity. "'Ordering"' and "'Symbols"' may be considered as syntactic sugar. Symbols replace well-defined nodes and hedges. Therefore, they improve reuse. Control flow in the validation process is of crucial importance.

From the application's perspective, the schema style is essential. If a schema language is pattern-styled, then even complex schemas with a difficult syntax description may easily be expressed. That relates to the regularity of schema languages.

W.r.t. to complexity, time and memory consumption play a key role. However, because of heterogeneous program systems, this criterion is often totally underestimated. Because of the prototypical Haskell implementation, a dynamic profile does not make sense. Both evaluation order, as well as eliminated

(a) from the perspective of a developer

Openness	Extensibility	Variability
Rules set Constraints	new Command Tags Modules/Namespaces	Command Language Instantiation Data

(b) according to the degree of objective consideration

Objective	Subjective
Language Type Complexity	Tool support Usability Self-explanation

TABLE XI
TAXONOMIES OF THE COMPARISON CRITERIA

bottlenecks, directly affect the efficiency of validators.

In addition to openness,structured rules may improve the comparison towards unification. Excluding constraints may cause contradiction towards unification and make the definition of new predicates error-prone and bloated. That is why an investigation of including and excluding specifications shall be done. The following points seem most promising regarding modularisation of template and schema: integration into namespaces, node inclusion, and exchangeable command languages. The taxonomies from tab.X (a) already cover the criteria from tab.XI (b).

C. Semantic

This section introduces features of the considered schema languages and one template language. Typing is here of outstanding interest in schema and command languages and the representation of functions, the use of constraint and error handling.

1) Typing: Each template node must be well-typed. It is a precondition for instantiation and validation. On the one hand, there are element and text nodes. On the other hand, there are command tags. Depending on the concrete command, those tags also return either an element node, a text node or may return a hedge containing both as an instance (cf. [1]). If this condition is fulfilled, then implicitly well-formedness of the instance is guaranteed. Access to attributes can be replaced as access to element nodes, as discussed earlier, and therefore does not require further investigation. Depending on a given schema language, typing may also include referential integrity.

Instantiation uses PHP functions, where validation does not. Validation is considered when validating the associated command tag type, so descriptions are reused in a unified approach.

Unification does not matter about command tags. The fewer possibilities exist to instantiate a template node; the faster validation can perform (see sect.VI-E).

In XSLT, type safety is guaranteed in XPath-expressions for command tags. The result is either a hedge, an element node or a boolean value (see [11], [67]). Numbers and dates are included in hedges whose children nodes have exactly one node and a list of `<book/>`-nodes as a hedge with a certain amount of element nodes. Similarly to XTL, singular types are used to include and boolean types for controlling the instantiation. The consideration of each command tag's typing is essential to validation because, for each command tag, a decision needs to be made with how many instance nodes it corresponds. In other words, this means the type of the inferred template node is compatible with the type of the instance node.

That raises whether a command retrieving text could not accidentally return a singular node or a hedge. However, the text's interpretation as element nodes would insist tag-brackets are generated at least in the output. However, this is not possible in XTL and XSLT's default configuration because special characters are treated as XML entities. Type safety on text output can be influenced in XSLT by the attribute `'disable-output-escaping'`. Even well-formedness is not guaranteed in XSLT because an upper-most root node does not need to exist. Because of this, XSLT is not appropriate for the unification of instantiation and validation. Despite this, XTL does guarantee type safety.

Guaranteeing referential integrity is essential, especially in XML databases. In XTL, most existing command tags can be adapted without prior configuration. However, unique values and foreign keys must be supported by the command language.

XSD has techniques to assure referential integrity. So, for instance, DTD and RelaxNG do not support keys. DTD supports only fundamental unique elements. However, there is no modularisation. Therefore, no separation and no distinction by element names are possible.

Information about keys and uniqueness does not hinder unification. However, they are only relevant to validation, but not to instantiation. Another approach to separate a schema is to hoist relationships over keys. In order to do that, schema nodes need to be referenced. That is done by XPath-expressions — in analogy to XSD. The key-referenced relationship remains free of redundant nodes, but it may also invalidate schema relations if path expressions become invalid.

From the standpoint of standard rules (see sect.VI-F2), the support for referential integrity is a violation of the unification of instantiation and validation. Hence, instantiation primary runs for a given document only once. However, validation runs the first time in order to localise identifiers and unique elements and attributes. The first run is then still needed in order to check uniqueness and keys.

2) Functions: Functions are a vital acceptance criterion for template languages. In JSP and ASP, small calculations can be performed by functions. A local and remote function call may be triggered. Template languages like XSLT define functions with arity greater equals zero by named templates (cf. [31]). The most popular schema languages do not have corresponding mechanisms. Functions require that among its values, there is a possibility to check each value has a correspondence to at least one instantiation data field. If this is the case, then this value is valid according to the function. Otherwise, the function value is not valid according to a schema node. In other words, functions must be invertible, and the instantiation of a template must be an isomorphic mapping [31].

Since this harsh restriction is too hard in practice, functions may not be validated in general. It means generalised functions prevent the unification. Restrictions must be imposed.

Functions may be not defined in XTL, which would not even be meaningful for the mentioned reason. Referential transparency disallows it. Only macros may be stored locally in the global symbol environment. That is why only functions within the command language may be used.

3) Constraints: Constraints may be required in both processes. During validation, constraints restrain nodes and text. So, specifications may be including or excluding. XTL, RelaxNG and DTD belong to schema languages with including specifications. XSD is a hybrid. XSD may allow contradicting specifications for which it is impossible to define a valid document. For instance:

```
<xsd:element name="top">
 <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="second" type="BBB2"
     minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:complexType name="AAA">
  <xsd:sequence>
  <xsd:element name="x" type="xsd:string"
     minOccurs="3"maxOccurs="3"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="BBB2">
  <xsd:complexContent>
    <xsd:restriction base="AAA">
      <xsd:sequence maxOccurs="1">
          <xsd:element name="x"
           type="xsd:string"
           minOccurs="3"
           maxOccurs="3"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

<second/> is not a valid instance, because 'BBB2' requires an empty content-model and requires an element node as child, so "'<second><x>1</x></second>"' is also invalid.

An example of constraints is the explicit null. AAA may not have child nodes within DTD in order to specify an element node. Children shall be specified EMPTY. The full element node corresponding will be <!ELEMENT AAA EMPTY>. In RelaxNG <empty/> has the same effect. In XSD the same is achieved by an empty sequence.

Constraints on elements are helpful to restrict. However, partially specified names can be inconvenient if names do no more differ (enough). In contrast to that, the restriction of child nodes is important. So, for example, a schema containing <a/> with a text node, followed by <b/> is specified in XTL as <a><xtl:text select="/a"/><b/>. Similar looks the specification for RelaxNG and DTD. The XSD is relatively long (see fig.12). Here, <a>#<b/> is not exactly expressed, where '#' denotes an arbitrary text node. The weakness of content models in XSD is their imprecise position for text and element nodes. So, <a><b/># and <a>#<b/># are accepted, even so this was not originally intended.

That is why including schemas is easier to check than excluding because validation has only to check whether both schema and instance nodes fulfil the same predicates. When excluding instance nodes in a schema, all listed specifications must be checked. Excluding schemas make unification more difficult because a fixed amount of excluding predicates needs to be considered.

On unification, attribute constraints may not specify attribute values any closer. For this reason, attribute names must be given in a complete form – the same as it is with element nodes. In contrast to elements, the exclusion of attributes in schema languages is weaker and has to obey weaker restrictions. So, in RelaxNG, within an 'except'-node 'nsName' may exclude a particular namespace. In XSD, only a specific namespace may either be included or excluded. On the contrary, arbitrary attributes without namespaces can be added by the 'anyAttribute'. It reminds the mixed content model, but is still different.

In XTL Tag 'attribute' allows specifying attributes arbitrarily. From this perspective, XTL offers the best usability.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/..">
 <xsd:element name="a">
  <xsd:complexType mixed="true">
   <xsd:sequence>
    <xsd:element name="b"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Fig. 12. XSD-Schema for a mixed Content-Model

Multiplicities are another possibility to restrict hedges. In XSD, any non-negative multiplicities are expressible by attributes. RelaxNG only allows the multiplicities '0..' and '1..*'.*

Multiplicities cannot be expressed dynamically but only by predetermined values and variables. It ensures that, for instance, in XSD, regular expressions can be composed (see sect.VI-E), and cycles do not depend on variables. In XTL, there is currently no option to repeat element nodes at any time. A fixed number of repetitions is allowed after all. That is why <a/><a/><a/> cannot be described by $L(a^3)$. Multiplicities in XTL are recommended for the same. Kleene's star operator 'unbound' may remain unnoticed on a definition because it is already covered by 'xtl:for-each'. Since multiplicities remain the properties on regularity untouched, unification gets another valid construct to be used.

4) Error Handling: Error handling strongly depends on the concrete automaton (see sect.VI-E). The handling of errors in XTL differs from the behaviour in XSD and RelaxNG. That is mainly due to the prototypical stage of XTL. XSD and RelaxNG issue an error message containing the error position. The schema is well-formed and suffices the syntactic requirements of the schema language. Although RelaxNG may also be simulated by a non-deterministic automaton (see sect.VI-E),

the output of all previous locally occurred errors is avoided by implementing an error stack. The automaton is a big help to schema developers but requires further resources. In XTL, there shall be an error stack too.

D. Syntax

In this section, syntactic features of schema languages are considered. The more features a schema language has, the more powerful it is – this assumption is false in general. Besides unification criteria, learnability is also being considered. The goal of this section is to find answers to the following questions:

1) ***Do linguistic features improve unification?***
   What can be done, if possible at all, in order to increase unification?
2) ***Can lingual features of other languages be expressed within XTL?***
   Apart from the question if it is possible in general, the question of complexity arises second and of type safety third (see sect.VI-C1). Does the other way round also function?
3) ***How significant are those lingual features in detail?***
4) ***Is XTL minimal w.r.t. lingual feature amount?*** *Are some template and schema features missing?*

XTL combines the advantages of XML-schema languages, especially the vast amount of available tools and APIs. One advantage, however, is still missing in RelaxNG and XTL: it is the fixed association between the document instance and schema. It should explicitly be placed in an XML processing instruction or the top element node, so the relationship does not get lost, primarily when several schemas are used. The information regarding embedded command languages should also be explicit. The `realm`*-attribute achieves this.*

1) Symbols: In schemas, it is often helpful to summarise hedges and reuse them at different locations. The insertion of a hedge is comparable to constant symbols because it is assigned only once and remains invariant afterwards. The invariance touches template and schema nodes. That is why invariant hedges are sometimes called "'symbols"' which can be expressed by constants. Symbols are visible everywhere within a given schema for all considered schema languages. Some template languages like XSLT version 2.0 and Prolog (cf. [31]) offer variable agreements, which may be used for memoisation and as a placeholder. These variables are not considered symbols because they are dynamic and therefore violate referential transparency.

Symbols are so-called Substitution groups in XSD, Definitions in RelaxNG, and Macros in XTL. Symbols in XSD may restrict or extend and always refer to element nodes. Symbols introduced in RelaxNG behave isomorphic to `xtl:call-macro` *and* `xtl:macro`*.*

In contrast to XSD in XTL and RelaxNG, text and element nodes may be appended before and after a macro call, where the ordering does not change (see sect.VI-D3). Symbols do not violate referential transparency due to determinism, so these are well-defined before instantiation and validation. That is why they harmonise both processes, instantiation and validation. Variables with mutable values on the other side hinder unification because these immediately influence validation (see sect.VI-C2).

The following coarse-grained structure would be namespaces. In XTL, namespaces are present, but because of their prototypical characteristics, they are not supported by XTL-implementations. In other schema languages, namespaces can either be within the top root node or RelaxNG or be local in RelaxNG in all children nodes. Apart from that, namespaces in RelaxNG and XTL can also be applied to attributes. The attribute `ns` *in element nodes does just that. Namespaces are modules too. By doing so, namespaces contribute to an ongoing unification of template language and schema languages.*

XSD has numerous simple and complex data types. In contrast to this, RelaxNG has only 2, and vocabularies of other schema languages are imported. XTL has only the data types from the previous section.

2) Control Flow: XML tags determine the control flow in schemas. The flow controls validation. Upon closer examination, `xtl:if` *and* `xtl:for-each` *can be interpreted as regular expressions. Simple conditions are represented by "'$r \mid \varepsilon$"', nested conditions with optional alternatives, and by a selection of "'$r_1 \mid r_2 \mid ... \mid \varepsilon$"', where '$r_j$' denote hedges of corresponding consequences.*

`xtl:if`*,* `xtl:for-each` *and* `xtl:call-macro` *determine the control flow in XTL, where the latter correspond to hedge substitutions (cf. sect.VI-C3). A matching* `xtl:for-each` *can substitute every* `xtl:if`*, but not vice versa. Hence,* `xtl:for-each` *is a universal element. In XSD, conditions may also be expressed by multiplicities with lower bound* `0` *and upper bound* `1`*.* `xtl:if`*-conditions can express multiple selections and nested conditions. These are the same from the standpoint of validation if the proposed right-associated OBDD is chosen (cf. sect.IV). So, a simple and a nested condition as well as multiple selections are preferred for unification. When it comes to instantiation, refined conditions are beneficial since unconsidered cases could be dropped, and the overall instantiation increased.*

3) Ordering: The ordering is of utmost importance to validation. Without an ordering, the increase of complexity for any validation algorithm could be tremendous (cf. sect.VI-E). Hence, an unspecified ordering requires testing for all possible permutations of all possible hedges. The amount of permutations is $n!$, where n is the number of children for a given hedge.

In XSD, the ordering can optionally be specified. By doing so, readability suffers and redundancy increases. A specification would be more efficient in terms of adequacy if only the provided ordering were allowed. The concrete ordering should always be taken out by expressions of a given schema language and should be done explicitly. Often certain permutations are not desired, and systematic exclusion of all unwanted

would critically increase complexity too. For example, in XSD, there exist different representations of "'<a/><b/>"'. This hedge may be represented by the unordered regular expression `(a|b)*`. *This expression is interpreted under a specific ordering mode and is expressed by multiplicities. So,*

```
<xsd:choice minOccurs="0"
maxOccurs="unbounded">r</xsd:choice>
```

describes the same tree language as:

```
<xsd:sequence minOccurs="0"
maxOccurs="unbounded">r</xsd:sequence>
```

The ordering would be restricted in RelaxNG by the tag 'notallowed' *to empty hedges only.*

In XTL and DTD, only those orderings are valid, matching the ordering specified in the schema. The implicit permutation of hedges is disallowed. In general, a permutable hedge '$e_0e_1...e_n$*' can be expressed as a regular expression*

$$\texttt{OR}\,(e_0 \cdot \overline{\textsf{permutate}}^{1}(e_1...e_n)) \;\ldots\; (e_n \cdot \overline{\textsf{permutate}}^{1}(e_1...e_{n-1}))$$

(see sect.IV).

This representation of $\overline{\textsf{permutate}}^{1}$ *is syntactic sugar and is believed to be safe due to its closeness. However, permutation would need to have a meaningful counterpart on instantiation when it comes to unification. If instantiated hedges ought to be permuted, then it can be stated that permutation increases unification w.r.t. instantiation and validation. A '*permutate'*-tag shall be introduced to XTL for the explicitly specified permutation.*

*Because of the described dramatic drop in performance on validation in general, unknown hedges and suffixes of known nodes are not considered currently in XTL and therefore are not implemented yet. Unknown prefixes can be expressed by '*include'*-tags and are guarded by '*xtl:for-each'*. The only severe disadvantage is an extensive search right at the beginning of the unknown prefix. It is a practical advantage to place known nodes at the start of a hedge and as close as possible to the document's beginning.*

4) Patterns: XSD is the most pattern-styled language among all considered schema languages (cf. [48]). In contrary, XTL, RelaxNG and DTD have a grammar-styled representation which is too less pattern-styled (see [32], [17]).

*Besides the style of a schema language, syntactic notation, e.g., using regular operator, greatly influences usability. In RelaxNG, the tag '*oneOrMore'* is used as a replacement for the plus-operator. Operators of the command language do not obey regularity criteria in XSD because restrictions on symbols (see sect.VI-D1) violate those. Expressions of the command language do not have any significance because they are ignored on validation.*

*5) Usability: Usability is worsened in XSD by redundant '*complexType'* definitions. Both XSD and RelaxNG lack adequate representations of element nodes. So,* `<xsd:element name="...">` *or* `<element> <name .../>...` *must be used as node constructors. Node definitions in DTD are not adequate. However, this cannot be resolved by any other means because of its non-XML notation. In contrast to this, XTL can take schema nodes exactly as they are. Specifications take nodes exactly as they are and simplify unification because template, schema, and instance nodes are all congruent.*

6) Syntactic Sugar: Syntactic sugar in this work's scope means command tags that can be removed from a given language, s.t. expressibility neither regarding schema language nor template language diminishes. The more features are shared among schema and template languages, the more unification increases because expressibility increases relatively.

Idioms and syntactic sugar have the following characteristics:

- ***Openness:** Idioms must be expressible just by some core functions.*
  Herewith, neither referential transparency may be violated, nor additional assumptions about instantiation data are allowed.
- ***Extension:** Idioms may not extend the expressibility of a schema language.*
  For example, this means that regular schema languages may naturally recognise ranked-competing fragments due to a weaker tag. Nevertheless, ranked-competing schema languages may not introduce tags, which would enable regular schema language recognition.

Openness causes both processes, validation and instantiation, are proceeded by another step is turning its product, the document, into a form free of sugared tags. Simple forms lead validators and schema simplifiers (see sect.VI-F2) to heavily reduced rules sets. For instance, in RelaxNG, this is called "'simple syntax"' [20].

An extension of a ranked-competing schema language to a regular schema language leads to most rules of a validator must be dropped since an increase in non-determinism leads immediately to further matching cases. Here, a validator is assumed to be fully described by a rule set. The inversion makes a conflict visible now since non-deterministic matching rules need to be excluded by the syntax.

It is compulsory syntactic sugar has unique semantics in both the template language and the schema language. If an idiom has only in one of both languages a non-empty semantic, then exclusions may only be hard to formulate and are less plausible.

E. Complexity

*A closer description of complexity is given in sect.IV. The validation algorithm is not bound by polynomial complexity w.r.t. schema length (see [1]) herewith. The complexity is caused by rules splitting string and '*Then'*-nodes non-deterministically. As a result of this, the potential search space for validation is drastically expanded. Nonetheless, this approach ends with a higher expressibility than XSD or DTD. The memory consumption is directed by runtime behaviour. When an alternative occurs, the last valid state before entering the recursion has to be stored. The XTL validation requires*

only the memory needed according to the maximal recursion depth. Solutions once dropped are not considered a second time. Only open solution not yet considered must be stored on lazy evaluation.

The separate handling of attributes does not cause a significant rise in complexity.

The infinite OBDDs proposed in sect.V have non-polynomial complexity on lazy evaluation, but only if fixpoints exist in the schema. From the practical perspective, infinite instance documents are disallowed because validation only considers enclosed documents. Instantiation can accept infinite OBDDs as input. Because in contrast to validation, instantiation does not require the entire document is present. Consecutive tags in a hedge may be instantiated independently (see sect.III-A2). Exceptions are macros having infinite bodies but whose overall structure is well-defined. In the case of finite templates without left-recursion, instantiation always terminates with a well-formed resulting instance document. The runtime complexity of instantiation is bound by a polynomial which degree is the maximum number of nested loops. However, this does not hold in general for macro calls.

Macros lead to a non-polynomial runtime behaviour, which may be compensated by lazy evaluation.

Depending on a tree language (see sect.III), a schema language automata may recognise different granularity levels. DTD is expressed by a single-typed grammar, where a ranked-competing grammar expresses XSD. The latter allows more freedom [48]. RelaxNG and XTL are generated by regular grammars and allow maximal expressibility. So, XTL allows defining arbitrary (but at most regular) sequences of nodes.

On the other side, there is the syntax to be considered. XTL has a minimalistic syntax since no element is sugared (see sect.III, IV). RelaxNG has a relatively tiny amount of features, for instance, in comparison to XSD. However, this does not affect expressibility. XSD has poor expressibility but lots of sugar, which quickly leads to hard to read documents.

By missing macros or definitions, schema validation always terminates in XSD. Only for XTL and RelaxNG closedness properties hold regarding intersection, union and complement. That is why those languages are perfect for extensions. The same expressibility class practically means transformations into each other are possible without any severe hinder. There may also be transformations from XSD and DTD to XTL and RelaxNG. Beware the other direction is not possible in general.

Top-down deterministic validators recognise schemas in DTD. Top-down non-deterministic validators recognise RelaxNG and XTL. However, XSD is generated by a single-typed tree grammar [48]. That is why a top-down deterministic validator recognises XSD.

F. Unification

This section unification is considered in general towards documents and rules sets.

1) Documents: In order to unify templates and schemas, the following restrictions are recommended:

1) ***Type Safety**. Both template and instance document must be XML. Existing command tags need to have a semantic, which does not depend on surrounding tags and is still distinct among other command tags. In conclusion, this means validation should have relatively a few possibilities only to validate against an instance node.*
2) ***Abstraction from the Command Language**. Expressions to be calculated should be enclosed in the document structure. It means, functions must be eliminated and command languages ignored during validation. Instantiation data must be hidden during validation. Instantiation data may not be in the schema, so access is granted by attribute entries in command tags.*
3) ***Self-Similarity**. The more significant similarities between schema and instance document are the more straightforward validation is to describe and the bigger the intersection of shared language features.*

2) Rules Set: The essential difference between instantiators and validators is the underlying algorithms. A validator matches nodes, where an instantiator substitutes nodes (see [57]). The instantiator may process nodes in parallel, where a validator processes an instance document sequentially.

Instantiators and validators are based on rules sets. Rules may express denotational semantics for instantiation and validation if those are of the kind: $p \rightarrow_{a_0,\ldots,a_n} q$. Here, 'p' denotes a premise. The premise of instantiation contains a template node, where the premise of validation contains an instance node. 'a_j' represent constraints. 'q' denotes the result of an instantiation or validation. The evaluation ordering of instantiation and validation is controlled by constraints (cf. [31]). Rule sets do not contain instantiation data. However, this is an obstacle, because validation iterates the instance document in parallel to the schema document. Instantiation does not allow an actual procedure. First, this is because instantiation data may be arbitrary heterogeneous structures, which cannot be compared with the template document in general. Second, any assumptions about the internal structure of instantiation data are strictly prohibited, as shown in sect.III.

An essential benefit of a rule-based semantic is the openness towards compilers and interpreters with a rigid structure (cf. [14]) – as is the case of denotational semantics in sect.IV and implementation in sect.V. So, new elements are comfortably inserted into or removed from existing rules. So, $am + n$ new cases are inserted to the rule set of a validator in case of insertion of a new tag, where the bipartite graph consists of at most $m \times n$ edges (cf. fig.7). It means the insertion of a new element has a linear complexity increase in conclusion (even the constant factor "'1"'). Despite this are bottom-up parsers, which often require a change in a considerable amount of rules. It is worth mentioning that $m+n$ cases could have further distinction, for instance, for the concatenation of (possibly empty) command tags (cf. sect.IV-C1).

In contrast to this, an instantiator requires just one rule

into the denotational semantics since there are no non-deterministic cases and a new command tag has the same behaviour as other tags.

The reuse of short and self-explanatory templates is simplified. In order to effectively reuse documents of a schema language, simplifications with the document structure are recommended. That is why rule-based simplifiers shall be developed, where weak regular schema languages are advantageous. For example, the schema $a^*(ba^*)^*$ shall be restructured to $(a|b)^*$ [23].

From a verbal perspective, commands hinder unification, which either supports instantiation or validation. However, their implementation implies the assumption that the other function does not make assumptions about the origin. On instantiation, e.g. XSLT-stylesheets are XSLT-templates, and regular text patterns on validation.

VII. Summary

This work examined how much template instantiation can narrow schema validation for XML documents in a unification attempt. First, instantiation and validation were formalised. Properties towards their practical meaning were probed, and implementation was developed. Requirements for unification were elaborated, and a comparison was made based on these results.

The semantics were formulated in different ways. On the one side, denotational semantics specified the programs' behaviour. On the other side, rules demonstrated introduced data models used and transformed. The tree automaton model was used for evaluation. Optimisation techniques were discussed. The formalisation made it more evident that instantiation is adequately represented as a term-rewriting system and validation as graph-matcher, also a rule and term-based system.

Both semantics showed that the rules set for both instantiation and validation could not entirely be unified. However, the reuse of simplified code simplifies unification.

The implementation allowed the unification of both processes on the document level. A comprehensive test suite guarantees the validity of all implementations. A stack for error should be integrated in future with a Java implementation. The regular data model was prototypically introduced in Java.

Analysis showed that XTL has regular grammar properties, except macros, which extend expressibility and violate specific closeness properties. The extension requires further research towards practical means.

Moreover, it was shown termination does not hold and should not hold in general. An explanation was given why filters and arrows are not best, especially when XTL will be variable and extensive. Recommendations for improvement were given.

Instantiation showed XTL is not as universally applicable as, for instance, XSLT. For instance, there is no possibility to define arbitrary functions in a schema, which could be used later. It was found the expressibility of XTL directly depends on the command language. Instantiators work deterministic, where validators do not by default. Here, parallel validation should be considered further.

It was found that it is advantageous to restrict unification. If a schema language assigns a type to each slot, then validation may be simplified quite considerable. Regular properties imply syntactic sugar can be defined without any change in the schema language's expressibility. In order to obtain the most flexibility, command languages require adaptations. The introduced rules for instantiation and validation have, in consequence, that changes can be done quickly. A practical possibility to beat non-determinism is the construction of automata as described and the restriction of a schema language's expressibility. It was noted, however, that there can be drawbacks here. First, Rabin-Scott's powerset construction may cost too high efforts for XML. Second, a restriction to single-typed or ranked-competitive is not a solution because closeness-properties are violated.

The comparison showed that potentially all generated instance documents significantly impacted the unification — much more than the expressibility of encapsulated queries. Comparison criteria were introduced regarding syntax and semantics. Comparisons were taken out and marked accordingly. Despite its colossal syntax definitions, XSD was found weaker than XTL or RelaxNG. XTL as template language is quite universal. Because of its universality it is possible, for instance, to define keys for referential integrity. Variable orderings of foreseen attributes shall be considered as syntactic sugar. It is recommended to define a rule-based simplifier for XTL-schemas because it is estimated simple schemas, and tools will raise the acceptance of XTL.

VIII. Glossary

`Abstraction`.
denotes an anonymous function in the λ-calculus.

`Ad-hoc polymorphism`.
similar to ↗ generic polymorphism, abstracts from and restricts a data type, e.g. by subclassing.

`Active Server Pages`.
a certain ↗ template-language.

`Ambiguity`.
non-determinism in grammars caused by overlapping rules.

`Application`.
denotes in terms of λ-calculus the application of a given term to an ↗ abstraction. Is equal to β-reduction. Applications with term and abstraction being the same are self-applicative.

`Arrow`.
generalised ↗ monad having functions as input.

`Arity`.
The amount of parameters a function has. A function with arity zero is a constant.

`Command Language`.
A language embedded in some ↗ template-language for accessing ↗ instantiation data by placeholder plugins. XPath is the command language for ↗ XSLT, where JXPath is a reference implementation.

`Ranked-Competing`.
↗ Regular tree grammar whose ↗ hedges are uniquely decomposable.

`Bypass Attribute`.
a ↗ XTL-attribute for the stepwise instantiation of a ↗ template.

`Call-by-need evaluation`.
a particular case of ↗ lazy evaluation on which intermediate results are not calculated twice.

`Constraints`.
denotes general restrictions. Constraints during ↗ instantiation and ↗ validation select and specify nodes with ↗ command tags. In ↗ XTL constraints are specified by "'select"'-expressions. Constraints may also be used to specify valid and invalid variable and function domains. In the context of databases constraints guarantee referential integrity.

`Content-Model`.
describes a ↗ hedge in ↗ XSD.

`Definitions`.
denote dedicated ↗ symbol nodes in ↗ RelaxNG.

`Denotational Semantics`.
also known as functional semantics, denotes the functional behaviour for a given program. It uses an universal language for its syntax, e.g. set operators or an abstract programming language, and defines a relation between syntactical constructs and their meaning. Denotational semantics abstract from a certain machine platform and focuses on calculating output for a given input.

`Derivatives`.
denote mappings from regular expressions to regular automata as proposed by Brzozowski and Glushkov.

`Document reconstruction`.
reconstructs unknown ↗ instantiation data. It is in contrast to ↗ validation, where instantiation data is known. Document reconstruction can be considered as an inverse operation to ↗ instantiation.

`Document Object Model`.
is a data model for a given XML document.

`Regular (Tree Grammar)`.
the most powerful of all considered ↗ regular tree grammars, which has no restriction.

`Endomorphism`.
mapping whose domain and codomain both denote the same set.

`Exhaustive Search`.
searches for a proper non-deterministic ↗validation.

`Lazy Evaluation`.
evaluation ordering which includes only those steps essential for obtaining the final result. Within the λ-calculus, it corresponds to the outermost term reduction.

`Filter`.
functions having a polymorphic ↗ type `a→[a]`.

`fixpoint`.
in geometry denotes a point that is fixed for a given mapping. Similarly, a fixpoint in templates and schemas is a synonym for conditions that in loops do no further change. In λ-calculi, a fixpoint denotes a condition with an invariant (state) in recursions. ↗ left-recursions lead to unreachable fixpoints during ↗ instantiation and ↗ validation.

`Functional`.
↗ higher-order functions.

`Higher-order Functions`.
functions consuming functions as input and output. Known list-↗functionals include fold-left '`foldl`', mapping '`map`' and filter '`filter`'.

`Generic Polymorphism`.
abstraction of a certain data-type without further restrictions, see ↗ ad-hoc polymorphism.

`Glushkov-Automaton`.
finite determined automaton which recognises regular expressions.

`Grammar Style`.
schema language whose syntax can be expressed well by a grammar — in contrast to ↗ pattern-like schema languages.

`Graph-Matcher`.
program, which checks if two graphs are identical.

`HaXML`.
Haskell-API for processing XML documents (also see ↗ HXT).

`Hedge`.
synonymous for a list of children nodes in an XML document.

`Homomorphism`.
mapping for which the following equation holds: the product of the codomains equals the codomain of its products. Homomorphism holds, for instance, for the addition of a residue class ring.

`HXT`.
Haskell-API for processing XML documents (also see ↗ HaXML).

`Instance`.
result of the ↗ instantiation.

`Instantiation`.
process turning a ↗ template with ↗ instantiation data into a ↗ instance.

`Instantiation data evaluator`.
part of the ↗ placeholder-plugin, which on request by the ↗ template engine issues ↗ instantiation data.

`Instantiation Data`.
denotes data sources, which are bound during an ↗ instantiation to ↗ slots.

`Interleaving (Matching Rule)`.
in terms of ↗ RelaxNG, denotes a non-deterministic matching rule.

`Interpretation`.
generally denotes the codomain of a function. ↗ Instances may be considered as an interpretation of a ↗ template.

`Java Server Pages`.
a ↗ template-language.

`Canonisation`.
recursively sorts all attributes within element nodes ascending by name.

`Kleene's star operator`.
denotes the star operator within regular expressions.

`Combinator`.
denotes an ↗ abstraction in the λ-calculus that has no free variables.

`Commando-Tag`.
denotes all tags in ↗ template-languages, which control the instantiation.

`Competing`.
Two non-terminals compete with each other if their sets of possible beginnings share at least one terminal.

`Left-recursion`.
causes the instantiation of a macro does not terminate.

`Literal`.
denotes text nodes and childless element nodes.

`Local Tree Grammar`.
is the weakest of all considered ↗ regular tree grammars, in which a terminal does not appear in any other rule.

`Macro`.
denotes in ↗ XTL a ↗ symbol.

`Tag`.
is for the distinction of text.

`Matcher`.
↗ Graph-Matcher.

`Memoisation`.
denotes the ↗ call-by-need evaluation in programming languages.

`Model-View-Controller`.
architectural principle for the separation of concerns between model, view and control.

`Monad`.
denotes in Haskell an algebraic data type. Lists with '[]' as neutral element represents a monad w.r.t. the associative operation '++'.

`Pattern Style`.
schema language whose syntax can be described by a regular expression – rather than ↗ in a grammar style.

`Non-deterministic Finite Automaton`.
automaton, which recognises regular expressions and has a finite number of states.

`Non-monotone and monotone operators`.
Non-monotone operators alter the structure of passed (fragments of) documents. Monotone operators keep passed documents as is or extend those.

`Non-strict Functions`.
functions, which accept indefinite data structures as argument and do terminate after a finite leap of time.

`OBDD`.
ordered binary decision tree.

`Partial Derivatives Algorithm`.
Successive construction of an NFA from a regular expression. Partial derivatives for a particular terminal-symbol are evaluated ↗ in call-by-need mode.

`Permutation`.
↗ commando-tag that swaps nodes.

`Path Problem`.
Theoretic questions regarding paths in a given graph.

`Placeholder-Plugin`.
a module evaluating specific requests formulated in a ↗ command language.

`Precedence (of an operator)`.
synonym for inverse operator priority. Operators with the lowest precedence have the highest priority over other operators.

`Processing-Instruction`.
XML-nodes containing a non-functional, but descriptive annotation to an XML node.

`Source Language`.
Formal Language ↗ instantiation data has to obey.

`Redex`.
denotes in λ-calculus terms, which may be reduced.

`Referential Transparency`.
A property that holds, when the evaluation of a function or an expression does not cause any side effects.

`Regular Tree Automaton`.
A tree automaton, that accepts regular ↗ tree languages.

`Regular Tree Grammar`.
A regular formal grammar whose terminal symbols extend to trees.

`Regular Tree Language`.
A formal language, which is generated by a ↗ regular tree.

`RelaxNG`.
XML-schema language, also see ↗ XSD.

`Schema`.
specifies an XML dialect for checking validity of an XML document.

`Transformation`.
Transforms a XML-document obeying one XML-schema to another XML-document possibly obeying another XML schema.

`Schematron`.
a XML schema language.

`Safe Commands`.
special ↗ commando-tags, which, when added to a certain schema, do not violate the closure property under union, intersection, minus. See also ↗ Syntactic Sugar.

`Single-Type Grammar`.
↗ Regular tree grammar whose non-terminals inside of ↗ hedges do not compete with each other.

`Slot`.
part of a ↗ template, which is substituted/filled during ↗instantiation.

`Slot-Markup Language`.
↗ template-language.

`Structured Query Language`.
the most popular standardised database query language.

`String-grammar`.
grammar whose derivations always generate words, also see ↗ word problem.

`Stylesheet`.
↗ XSLT-↗ template-document.

`Substitution group`.
denotes a ↗ symbol in ↗ XSD.

`Super-combinator`.
A specialised ↗ combinator, with all containing ↗ abstractions being super-combinator again. Combinators in imperative/procedural languages lead to modular programs. The introduction of new bindings, for example, turns $\lambda w.(\lambda x.xw)$ into $\lambda w.(\lambda xy.xy)w$.

`Symbol`.
synonym for a symbol binding to a ↗ hedge.

`Syntactic Sugar`.
Constructs or idioms of a programming language improve usability, but they can be replaced by more complicated constructs from the same language. Sugar does not extend functionality. It may only increase expressibility. Idioms worsening the expressibility of a language are called Syntactic Salt.

`Template`.
a XML-document containing ↗ slots. It is used for ↗ instantiation and obeys the rules of a ↗ template-language.

`Template Engine`.
Software, which instantiates ↗ templates.

`Template-Expansion`.
↗ instantiation.

`Template-Language`.
is described by ↗ command-tags, text nodes, element nodes.

`Term-Evaluator`.
Part of ↗ placeholder-plugins providing the ↗ template engine with formatted instantiation data.

`Tracing`.
tracking down errors by using a stack.

`Type`.
denotes a constraints on input and output parameters of a function in Haskell.

`Typing Problem`.
Theoretical question, if for a given λ.term e a type t may be inferred, s.t. $e :: t$.

`Type Isomorphism`.
Equality of two ↗ types, allowing only renaming of type variables.

`Type Constructors`.
serve in Haskell for algebraic data type definitions.

`Validation`.
checks whether for a given ↗ template instantiation with previously unknown ↗ instantiation data returns a document obeying a given schema.

`Vocabulary`.
domain of valid XML namespaces.

`Word Problem`.
Theoretical question, if a given word is element of a formal language generated by a ↗ String-grammar.

`XML-Entity`.
denotes a special character in XML.

`JXPath`.
an XPath-reference implementation for Java.

`XSD`.
a XML-schema language, also see ↗ RelaxNG.

`XSLT`.
is a XML ↗ template-language.

`Target Language`.
denotes the language of all ↗ instance documents obtained after ↗ instantiation.

`Membership-Function`.
denotes a discrete mapping between domain and a real value between 0 and 1.

IX. APPENDIX A: DENOTATIONAL SEMANTICS

Instantiation

Source: `https://rhaber123.github-.io/web-page/`

(S) $\mathcal{E}^{Start}[\![\underline{x}]\!]\underline{s}\pi := \mathcal{E}[\![\underline{x2}]\!]\underline{s}\pi$ *for* $\underline{x2} = \mathcal{E}^{r}[\![\underline{x}]\!]$

(E) $\mathcal{E}^{r}[\![$`ElX n a c`$]\!] :=$
let $\underline{attDefs} = \overline{filter}^{2}(\lambda\underline{child}.\mathcal{E}^{MA}[\![\underline{child}]\!])\underline{c}$,
$\underline{nodes} = \overline{filter}^{2}(\lambda\underline{child}.\overline{not}^{1}\ \mathcal{E}^{MA}[\![\underline{child}]\!])\underline{c}$
in `ElX n` $(\overline{qSort}^{1}(\underline{a} +\!\!+ \underline{attDefs}))$ $\underline{nodes}$

(I1) $\mathcal{E}[\![$`XTxt t`$]\!]\_\ \_ :=$ `XTxt t`

(I2) $\mathcal{E}[\![$`XAtt n v`$]\!]\_\ \_ :=$ `XAtt n v`

(I3) $\mathcal{E}[\![$`ElX n a c`$]\!]\underline{s}\pi :=$
let $\underline{mdefs} = \overline{filter}^{2}(\lambda\underline{child}.\mathcal{E}^{MM}[\![\underline{child}]\!])\underline{c}$,
$\underline{nodes} = \overline{filter}^{2}(\lambda\underline{child}.\overline{not}^{1}\ \mathcal{E}^{MM}[\![\underline{child}]\!])\underline{c}$
in `ElX n a`
$(\overline{concatMap}^{2}(\lambda\underline{node}.\mathcal{E}^{\alpha}[\![\underline{node}]\!](\underline{s},\underline{mdefs},\pi))$
$\underline{nodes})$

(A1) $\mathcal{E}^{\alpha}[\![$`XIf x gc`$]\!](\underline{s},\mu,(\vec{f}^{1}_{0},\vec{f}^{2}_{0},\vec{f}^{3}_{0},\vec{f}^{4}_{0})) :=$
if $(\vec{f}^{3}_{0}\underline{xs})$ *then*
$\overline{concatMap}^{2}(\lambda\underline{c}.\mathcal{E}^{\alpha}[\![\underline{c}]\!](\underline{s},\mu,(\vec{f}^{1}_{0},\vec{f}^{2}_{0},\vec{f}^{3}_{0},\vec{f}^{4}_{0})))\underline{gc}$
else `[ ]`

(A2) $\mathcal{E}^{\alpha}[\![$`XForEach x gc`$]\!](\underline{s},\mu,(\vec{f}^{1}_{0},\vec{f}^{2}_{0},\vec{f}^{3}_{0},\vec{f}^{4}_{0})) :=$
let $\underline{sels} = \vec{f}^{2}_{0}\underline{x}\ \underline{s}$
in $\overline{concatMap}^{2}\ (\overline{f3}^{1})\ \underline{sels}$
for $\overline{f3}^{1} = \lambda\underline{c}.$
$\overline{concatMap}^{2}(\lambda\underline{c2}.\mathcal{E}^{\alpha}[\![\underline{c2}]\!](\underline{c},\mu,(\vec{f}^{1}_{0},\vec{f}^{2}_{0},\vec{f}^{3}_{0},\vec{f}^{4}_{0})))\underline{gc}$

(A3) $\mathcal{E}^{\alpha}[\![$`XCallMacro m1`$]\!](\underline{s},\mu,\pi) :=$
let $\mu_2 = \overline{getMacro}^{1}\mu$
in $\overline{concatMap}^{2}(\lambda\underline{m}.\mathcal{E}^{\alpha}[\![\underline{m}]\!](\underline{s},\mu,\pi))\ \mu_2$
where `getMacro [ ] = []`
`getMacro (XCallMacro` $\underline{m2}$ $\underline{c}$`):`$\underline{xs}$
`|` $(\underline{m1} == \underline{m2})$ `=` $\underline{c}$
otherwise `getMacro xs`

(A4) $\mathcal{E}^{\alpha}[\![$`XTxt t`$]\!](\underline{s},\mu,(\vec{f}^{1}_{0},\_,\_,\_)) :=$
`[ XTxt` $\vec{f}^{1}_{0}$ $\underline{t}$ $\underline{s}$ `]`

(A5) $\mathcal{E}^{\alpha}[\![$`XInclude x`$]\!](\underline{s},\mu,(\_,\_,\_,\vec{f}^{4}_{0})) :=$
`[` $\vec{f}^{4}_{0}$ $\underline{x}$ $\underline{s}$ `]`

(A6) $\mathcal{E}^{\alpha}[\![$`ElX n a c`$]\!](\underline{s},\mu,\pi) :=$
`[ ElX n a`
$(\overline{concatMap}^{2}(\lambda\underline{child}.\mathcal{E}^{\alpha}[\![\underline{child}]\!](\underline{s},\mu,\pi))\underline{c})$
`]`

(A7) $\mathcal{E}^{\alpha}[\![$`TxtX t`$]\!](\_,\_,\_) :=$ `[ TxtX t ]`

Validation

(S) $\mathcal{E}^{Start}[\![\underline{x}]\!]\underline{s}\pi := \mathcal{E}[\![\underline{x2}]\!]\underline{s}\pi$ *for* $\underline{x2} = \mathcal{E}^{r}[\![\underline{x}]\!]$

(E1) $\mathcal{E}[\![$`Epsilon, TxtR ""`$]\!]\mu :=$ `True`
(E2) $\mathcal{E}[\![$`Epsilon, TxtR _`$]\!]\mu :=$ `False`
(E3) $\mathcal{E}[\![$`Epsilon, Epsilon`$]\!]\mu :=$ `True`
(E4) $\mathcal{E}[\![$`Epsilon, ElR _ _ _`$]\!]\mu :=$ `False`
(E5) $\mathcal{E}[\![$`Epsilon, Star _`$]\!]\mu :=$ `True`
(E6) $\mathcal{E}[\![$`Epsilon, TextR _`$]\!]\mu :=$ `True`
(E7) $\mathcal{E}[\![$`Epsilon, Then` $\underline{r1}$ $\underline{r2}$`]`$\!]\mu :=$
$\mathcal{E}[\![$`Epsilon,` $\underline{r1}$`]`$\!]\mu \wedge \mathcal{E}[\![$`Epsilon,` $\underline{r2}$`]`$\!]\mu$

(Then1) $\mathcal{E}[\![$`Then _ _, Epsilon`$]\!]\mu :=$ `False`

(Then2) $\mathcal{E}[\![$`Then` $\underline{r1}$ $\underline{r2}$`, TxtR` $\underline{text}$`]`$\!]\mu :=$
$\mathcal{E}[\![\underline{r1}$`, TxtR` $\underline{text}$`]`$\!]\mu$
$\wedge\ \mathcal{E}[\![\underline{r2}$`, Epsilon`$]\!]\mu$

(Then3) $\mathcal{E}[\![$`Then`
`(ElR` $\underline{name1}$ $\underline{atts1}$ $\underline{r1}$`)`
$\underline{r}$,
`ElR` $\underline{name2}$ $\underline{atts2}$ $\underline{r2}$
$\mathcal{E}[\![$ `ElR` $\underline{name1}$ $\underline{atts1}$ $\underline{r1}$`,` $]\!]\mu$
`ElR` $\underline{name2}$ $\underline{atts2}$ $\underline{r2}$
$\wedge\ \mathcal{E}[\![\underline{r}$`, Epsilon`$]\!]\mu$

(Then4) $\mathcal{E}[\![$`Then _ _, ElR _ _ _`$]\!]\mu :=$ `False`

(Then5) $\mathcal{E}[\![$`Then` $\underline{r1}$ $\underline{r2}$`, Star` $\underline{s}$`]`$\!]\mu :=$
$\vee$`[True|`
`(`$\underline{s1}$`,`$\underline{s2}$`)`$\leftarrow \overline{frontSplits}^{1}$`(Then` $\underline{r1}$ $\underline{r2}$`)`
$\wedge\ \mathcal{E}[\![\underline{s1}$`,` $\underline{s}$`]`$\!]\mu$
$\wedge\ \mathcal{E}[\![\underline{s2}$`, Star` $\underline{s}$`]`$\!]\mu$ `]`

(Then6) $\mathcal{E}[\![$`Then` $\underline{r1}$ `Epsilon,` $]\!]\mu :=$
`Then` $\underline{s1}$ $\underline{s2}$
$\mathcal{E}[\![\underline{r1}$`, Then` $\underline{s1}$ $\underline{s2}$`]`$\!]\mu$

(Then 7) $\mathcal{E}[\![$`Then` $\underline{r1}$ $\underline{r2}$`, Then` $\underline{s1}$ $\underline{s2}$`]`$\!]\mu :=$
$\vee$`[True|`
`(`$\underline{t1}$`,`$\underline{t2}$`)`$\leftarrow \overline{splits}^{1}$`(Then` $\underline{r1}$ $\underline{r2}$`)`
$\wedge\ \mathcal{E}[\![\underline{t1}$`,` $\underline{s1}$`]`$\!]\mu\ \wedge\ \mathcal{E}[\![\underline{t2}$`,` $\underline{s2}$`]`$\!]\mu$`]`

(Then 8) $\mathcal{E}[\![$ `Then (TxtR _) Epsilon,` $]\!]\mu :=$
`TextR _`
`True`

(Then 9) $\mathcal{E}[\![$`Then _ _, TextR _`$]\!]\mu :=$
`False`

(Φ) $\mathcal{E}[\![\underline{inst}$`, MacroR` $\underline{mname}$`]`$\!]\mu :=$
$\mathcal{E}[\![\underline{inst}$`,`$\underline{word}$`]`$\!]\mu$
for $\underline{word} = \overline{getMacro}^{2}$ `mname` μ

(Ω) $\mathcal{E}[\![\underline{inst}$`, Or` $\underline{r1}$ $\underline{r2}$`]`$\!]\mu :=$
$\mathcal{E}[\![\underline{inst}$`,` $\underline{r1}$`]`$\!]\mu \vee \mathcal{E}[\![\underline{inst}$`,` $\underline{r2}$`]`$\!]\mu$

```
(#1)  E[[TxtR text, Then r1 r2]]μ :=
        ∨[ True |
        (s1, s2) ← splitText^1 text
        ∧ E[[TxtR s1, r1]]μ
        ∧ E[[TxtR s2, r2]]μ ]
(#2)  E[[TxtR "", Epsilon]]μ := True
(#3)  E[[TxtR _, Epsilon]]μ := False
(#4)  E[[TxtR "", Star _]]μ := True
(#5)  E[[TxtR text, Star r]]μ :=
        if (∨[ True |
        (s1, s2) ← frontSplitText^1 text
        ∧ E[[TxtR s1, r]]μ
        ∧ E[[TxtR s2, Star r]] == True)
        then True
        else E[[TxtR text, Epsilon]]μ
(#6)  E[[TxtR text, TextR _]]μ := True
(#7)  E[[TxtR text1, TxtR text2]]μ :=
        text1 == text2
(#8)  E[[TxtR text, ElR _ _ _]]μ := False

(ElR1)  E[[ElR name1 atts1 r1, ]]μ
        :=  ElR name2 atts2 r2
            (name1 == name2)
            ∧ (qSort^1 atts1 == atts3)
            ∧ E[[r1, r3]]μ
            for (ElR _ atts3 r3) =
                extractAttributes^1
                (ElR name2 atts2 r2)
(ElR2)  E[[ElR name1 atts1 r1,  ]]μ:=
          Then (ElR name2 atts2
          r2) s
            E[[ElR name1 atts1 r1,  ]]μ
              ElR name2 atts2 r2
            ∧ E[[Epsilon, s]]μ
(ElR3)  E[[ElR name1 atts1 r1,  ]]μ:=
          Then (Or s1 s2) s
            (E[[ElR name1 atts1 r1,  ]]μ
              Or s1 s2
            ∧ E[[Epsilon, s]]μ)
            ∨ (E[[Epsilon, Or s1 s2]]μ
            ∧ E[[ElR name1 atts1 r1, s]]μ)
(ElR 4)  E[[ElR name1 atts1 r1,  ]]μ:=
           Then (Star s1) s
             (E[[ElR name1 atts1 r1, s1 ]]μ
             ∧ E[[Epsilon, s]]μ) ∨
             E[[ElR name1 atts1 r1, s]]μ
(ElR 5)  E[[ElR name1 atts1 r1,  ]]μ:=
           Then (MacroR m) s
             (E[[ElR name1 atts1 r1,  ]]μ
               MacroR m
             ∧ E[[Epsilon, s]]μ) ∨
             (E[[Epsilon, MacroR m]]μ
             ∧ E[[ElR name1 atts1 r1, s]]μ)
(ElR6)  E[[ElR _ _ _, Then _ _]]μ := False
(ElR7)  E[[ElR name atts r, Star s]]μ:=
          E[[ElR name atts r, s]]
(ElR8)  E[[ElR _ _ _, TextR _]]μ := False
(ElR9)  E[[ElR _ _ _, Epsilon]]μ := False
(ElR10)  E[[ElR _ _ _, TxtR _]]μ := False
```

X. APPENDIX B: PARTIAL-DERIVATIVES ALGORITHM

(Source: [4])

Given: Regular expression $t = x^* \cdot \underbrace{(x \cdot x + y)^*}_{r}$

To be found: NFA with $L(t)$*?*

Step 1: Determine linear form

$$
\begin{aligned}
lf(t) &= \underline{lf(x^*)} \odot r \cup lf(r) \\
&= \underline{(lf(x) \odot x^*)} \odot r \cup lf(r) \\
&= \underline{(\{< x, \lambda >\} \odot x^*)} \odot r \cup lf(r) \\
&= \underline{(\{< x, x^* >\})} \odot r \cup lf(r) \\
&= \underline{\{< x, x^* \cdot r >\}} \cup lf(r) \\
&= \{< x, t >\} \cup lf(r) \\
&= \{< x, t >, < x, x \cdot r >, < y, r >\}
\end{aligned}
$$

$$
\begin{aligned}
lf(r) &= \underline{lf(x \cdot x + y)} \odot r \\
&= \underline{(lf(x \cdot x) \cup lf(y))} \odot r \\
&= \underline{((lf(x) \odot x) \cup lf(y))} \odot r \\
&= \underline{((lf(x) \odot x) \cup \{< y, \lambda >\})} \odot r \\
&= \underline{((< x, \lambda > \odot x) \cup \{< y, \lambda >\})} \odot r \\
&= \underline{(\{< x, x >\} \cup \{< y, \lambda >\})} \odot r \\
&= \underline{\{< x, x >, < y, \lambda >\}} \odot r \\
&= \{< x, x \cdot r >, < y, r >\}
\end{aligned}
$$

$$
\begin{aligned}
lf(x \cdot r) &= \underline{lf(x)} \odot r \\
&= \underline{\{< x, \lambda >\}} \odot r \\
&= \{< x, r >\}
\end{aligned}
$$

All linear forms are determined now for the second component.

Step 2: Apply the Partial-Derivatives algorithm:

$$
\begin{aligned}
< PD_0, \Delta_0, \tau_0 > &:= < \emptyset, \{t\}, \emptyset > \\
PD_1 &:= PD_0 \cup \Delta_0 = \{t\} \\
\Delta_1 &:= \cup_{p \in \Delta_0} \{q \mid < x, q > \in lf(p) \ \wedge q \notin PD_1\} = \{x \cdot r, r\} \\
\tau_1 &:= \tau_0 \cup \{< p, x, q > | p \in \Delta_0 \wedge < x, q > \in lf(p)\} \\
&= \{< t, x, t >, < t, x, x \cdot r > , < t, y, r >\}
\end{aligned}
$$

$$
\begin{aligned}
< PD_1, \Delta_1, \tau_1 > &:= < \{t\}, \{x \cdot r, r\}, \{< t, x, t >, < t, x, x \cdot r >, < t, y, r >\} > \\
PD_2 &= \{t, x \cdot r, r\} \\
\Delta_2 &= \emptyset \\
\tau_2 &= \{< t, x, t >, < t, x, x \cdot r > , < t, y, r >, < x \cdot r, x, r > , < r, x, x \cdot r >, \\
&\quad < r, y, r >\}
\end{aligned}
$$

$$
\begin{array}{lcl}
< PD_2, \Delta_2, \tau_2 > & := & < \{t, x \cdot r, r\}, \emptyset, \tau_2 > \\
PD_3 & = & \{t, x \cdot r, r\} \\
\Delta_3 & = & \emptyset \\
\tau_3 & = & \tau_2 \rightarrow Halt!
\end{array}
$$

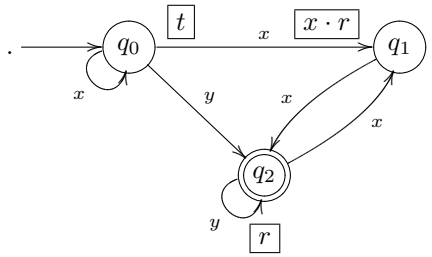
Final states:

$$F \subseteq PD_3 := \{f | f \in PD_3 \wedge f \in \tau_1\} = \{r\}$$

Remaining states:

$$PD_3 \setminus F = \{t, x \cdot r\}$$

Step 3: Building NFA:



τ_{Reg0} .. *terms that do not contain* ε
τ_{Reg1} .. *terms that do contain* ε

REFERENCES

[1] *Frank Atanassow and Johan Jeuring. Customizing an XML-Haskell data binding with type isomorphism inference in Generic Haskell. Sci. Comput. Program., 65(2):72–107, 2007.*

[2] *Lloyd Allison. A practical introduction to denotational semantics. Cambridge University Press, 1988.*

[3] *Krasimir Angelov and Simon Marlow. Visual Haskell Version 0.2, Download vom 01.01.2007. http://www.haskell.org/visualhaskell/.*

[4] *V. Antimirov.* Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science, 155(2):291–319, 1996.*

[5] *H. Barendregt.* The Lambda Calculus: its Syntax and Semantics. *No.103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1981.*

[6] *A. Boldakov, M. Grinev.* Transformation of XML Data Using Updates without Side Effects. *Programming and Computer Software, Vol. 32, No.5:255–267, 2006.*

[7] *P.V. Biron, K. Permanente, and A. Malhotra.* XML Schema Part 2: Datatypes Second Edition, *W3C Recommendation 28 October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.*

[8] *A. Balmin, Y. Papakonstantinou, and V. Vianu.* Incremental Validation of XML documents. *ACM Trans. Database Syst., 29(4):710–751, 2004.*

[9] *Janusz A. Brzozowski.* Derivatives of regular expressions. *J. Assoc. Comput. Mach., 11(4):481–494, 1964.*

[10] *B.I.-Wissenschaftsverlag.* Duden Informatik. *Dudenverlag, Mannheim, 1993.*

[11] *J. Clark, S. DeRose.* XML Path Language (XPath), *W3C Recommendation, 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.*

[12] *H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi.* Tree Automata Techniques and Applications, *release October 1997, 1st 2002. http://www.grappa.univ-lille3.fr/tata.*

[13] *K. Czarnecki, U.W. Eisenecker.* Components and generative programming *(invited paper), 1999.*

[14] *K. Czarnecki, U.W. Eisenecker.* Generative Programming: Methods, Tools, and Applications. *Addison-Wesley, 2000.*

[15] *E. Clarke, O. Grumberg, and D. Peled.* Model Checking. *MIT Press, 2000.*

[16] *B. Chidlovskii.* Using Regular Tree Automata as XML Schemas. *In ADL '00: Proc. of the IEEE Advances in Digital Libraries 2000, p.89, Washington, DC, USA, 2000. IEEE Computer Society.*

[17] *J. Clark.* An algorithm for RELAX NG validation, *download form 13.01.2007. http://www.thaiopensource.com/relaxng/derivative.html.*

[18] *J. Clark.* TREX – Tree Regular Expressions for XML, *download from 13.01.2007. http://www.thaiopensource.com/trex/.*

[19] *J. Clark.* XSL Transformations (XSLT) Version 1.0, W3C Recommendation. *16th November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.*

[20] *J. Clark, M. Makoto.* Relax NG Specification, *Committee Specification from 3 December 2001. http://www.oasis-open.org/committees/relax-ng/.*

[21] *M.D. Coen.* Interactive program derivation. *Technical report, University of Cambridge, Computer Laboratory, November 1992.*

[22] *L. Cardelli, P. Wegner.* On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys, 17(4):471–522, 1985.*

[23] *D.Z. Du, K.-I. Ko.* Problem Solving in Automata, Languages, and Complexity. *Wiley, 2001.*

[24] *E. Denti, A. Omicini, and A. Ricci.* Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program., 57(2):217–250, 2005.*

[25] *M. Erwig, D. Ren.* Monadification of functional programs. *Sci. Comput. Program., 52(1-3):101–129, 2004.*

[26] *M. Fitzgerald.* Validation by Instance — Translating the DTD to RELAX NG, *download from 13.01.2007. http://www.xml.com/pub/a/2002/08/28/validation.html?page=last.*

[27] *M. Fowler.* Refactoring: Improving the Design of Existing Code. *Addison-Wesley, 2000.*

[28] *D.C. Fallside, P. Walmsley.* XML Schema Part 0: Primer Second Edition, *W3C Recommendation 28 October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.*

[29] *Glasgow Haskell Compiler. http://www.haskell.org/ghc.*

[30] *D. Grune, C. Jacobs.* Parsing Techniques: A Practical Guide. *Ellis Horwood, 1991.*

[31] *R. Haberland.* Transformation von XML-Dokumenten mittels Prolog (unveröffentlicht). *Technische Universität Dresden, Oktober 2006.*

[32] *F. Hartmann.* XML Template Language Specification 7.0 *from 01.01.2007 (unpublished).*

[33] *F. Hartmann.* An Architecture for an XML-Template Engine Enabling Safe Authoring. *In DEXA '06: Proc. of the 17th Intl. Conf. on Database and Expert Systems Applications, pp.502–507, Washington, DC, USA, 2006. IEEE Computer Society.*

[34] *P. Hudak, A. Courtney, H. Nilsson, and J. Peterson.* Arrows, robots, and functional reactive programming, *2002. http://citeseer.ist.psu.edu/hudak02arrows.html.*

[35] *D. Herington.* HUnit 1.0 — Haskell Unit Testing, *download from 01.01.2007. http://hunit.sourceforge.net/.*

[36] *H. Hosoya, A. Frisch, and G. Castagna.* Parametric polymorphism for XML. *ACM SIGPLAN Not., 40(1):50–62, 2005.*

[37] *J. Hughes.* Functional programming languages and computer architecture. *5th ACM conference, Cambridge, MA, USA, August 26-30, Springer, 1991.*

[38] *J. Hughes.* Generalising monads to arrows. *Sci. Comput. Program., 37(1-3):67–111, 2000.*

[39] *J. Hughes.* Programming with Arrows, *Book Series Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, 2005.*

[40] *JXPath Jakarta Project. http://jakarta.apache.org/commons/jxpath/.*

[41] *M.H. Kay.* DTDGenerator — A tool to generate XML DTDs, *download from 13.01.2007. http://saxon.sourceforge.net/dtdgen.html.*

[42] *J. Kerievsky.* Refactoring to Patterns. *Springer, 2006.*

[43] *M. Kenji, S. Hiroyuki.* Static optimization of XSLT stylesheets: template instantiation optimization and lazy XML parsing. *In DocEng '05: Proc. of the 2005 ACM symposium on Document engineering, pp.55–57, NY, USA, 2005. ACM Press.*

[44] *T. Kuseler.* Design und Entwicklung eines RelaxNG Schema Validators auf Basis der Haskell XML Toolbox. *Diplomarbeit, Fachhochschule Wedel, 2005. http://www.fh-wedel.de/ si/HXmlToolbox/index.html.*

[45] *R. Lämmel.* Reuse by Program Transformation. *In Greg Michaelson and Phil Trinder, editors, Functional Programming Trends Workshop. Intellect, 2000.*

[46] *D. Lee, W.W. Chu.* Comparative analysis of six XML schema languages, *SIGMOD Record (ACM Special Interest Group on Management of Data), 29(3):76–87, 2000. http://citeseer.ist.psu.edu/lee00comparative.html.*

[47] *S. Marlow.* Haddock project — A Haskell Documentation Tool *http://www.haskell.org/haddock.*

[48] *M. Murata, D. Lee, and M. Mani.* Taxonomy of XML Schema Languages using Formal Language Theory. *In Extreme Markup Languages, Montreal, Canada, 2001. http://citeseer.ist.psu.edu/murata00taxonomy.html.*

[49] *M. Murata.* Forest-regular languages and tree-regular languages*, 1995.*

[50] *M. Murata.* Hedge Automata: a Formal Model for XML Schemata*. Web page, 1999.*

[51] *M.L. Noga, S. Schott, and W. Löwe.* Lazy XML processing. *In DocEng'02: Proc. of the 2002 ACM symposium on Document engineering, pp.88–94, NY, USA, 2002. ACM Press.*

[52] *T. Parr.* Enforcing strict model-view separation in template engines*, 2004. http://citeseer.csail.mit.edu/parr04enforcing.html.*

[53] *T. Parr.* A Functional Language For Generating Structured Text*, 2006. http://www.cs.usfca.edu/ parrt/papers/ST.pdf.*

[54] *A.C. Pereira, F. Hartmann, and K. Kadner.* A distributed staged architecture for multimodal applications (extended abstract)*. In Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105, Kollen Verlag, Bonn, March 2007.*

[55] *S.L. Peyton-Jones.* The implementation of functional programming languages*. Prentice-Hall Intl. Series in Computer Science. Prentice Hall, 1987.*

[56] *S.L. Peyton-Jones, D. Lester.* Implementing Functional Languages: A Tutorial*. Intl. Series in Computer Science. Prentice-Hall, 1992.*

[57] *E. Rahm, P.A. Bernstein.* A survey of approaches to automatic schema matching*. VLDB J., 10(4):334–350, 2001.*

[58] *Uwe Schmidt.* Haskell XML Toolbox 7.1*, download from 01.01.2007, http://www.fh-wedel.de/šsi/HXmlToolbox/.*

[59] *SNOW: Services for Nomadic Workers. http://www.snow-project.org.*

[60] *H.S. Thompson, D. Beech, and M. Maloney.* Xml schema part 1: Structures second edition, W3C recommendation*, 28 october 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.*

[61] *R.D. Tennent.* The denotational semantics of programming languages*. Commun. ACM, 19(8):437–453, 1976.*

[62] *S. Thompson.* Haskell: The Craft of Functional Programming*. Int. Comp. Sci. Addison Wesley, second edition, 1999.*

[63] *S. Thompson.* Regular Expressions and Automata using Haskell*. Technical report, University of Kent at Canterbury, 2000.*

[64] *S. Thompson, C. Reinke.* Refactoring Functional Programs*. Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001. http://www.cs.kent.ac.uk/pubs/2001/1334.*

[65] Why Haskell matters? – HaskellWiki*, download from 13.01.2007, http://www.haskell.org/haskellwiki/Why Haskell matters.*

[66] *P. Wadler.* Monads for functional programming. In Advanced Functional Programming*, 1st Intl. Spring School on Advanced Functional Programming Techniques-Tutorial Text, pp.24–52, London, UK, 1995. Springer.*

[67] *P. Wadler.* A formal semantics of patterns in XSLT and XPath. Markup Languages: Theory and Practice*, 2:183–202, 2000.*

[68] *M. Wallace. Haskell and XML. download from 13.01.2007 http://www.cs.york.ac.uk/fp/HaXml/.*

[69] *M. Wallace, C. Runciman.* Haskell and XML: Generic Combinators or Type-Based Translation?*, ACM SIGPLAN Not., 34(9):148–159, 1999.*

[70] *Apache Xerces. http://xerces.apache.org/.*