Dual Syntax for XML Languages

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach

Department of Computer Science, University of Aarhus, Denmark
{brabrand,amoeller,mis}@brics.dk

Abstract

XML is successful as a machine processable data interchange format, but it is often too verbose for human use. For this reason, many XML languages permit an alternative more legible non-XML syntax. XSLT stylesheets are often used to convert from the XML syntax to the alternative syntax; however, such transformations are not reversible since no general tool exists to automatically parse the alternative syntax back into XML.

We present *XSugar*, which makes it possible to manage dual syntax for XML languages. An XSugar specification is built around a context-free grammar that unifies the two syntaxes of a language. Given such a specification, the XSugar tool can translate from alternative syntax to XML and vice versa. Moreover, the tool statically checks that the transformations are reversible and that all XML documents generated from the alternative syntax are valid according to a given XML schema.

Key words: XML transformation, non-XML syntax, reversible, bidirectional

1 Introduction

XML has proven successful as a machine processable data interchange format. There exist numerous APIs for processing XML data in general purpose programming languages and also many specialized XML processing languages, such as XSLT and XQuery. Realizing the benefits of using XML, an increasing number of new languages, ranging from loosely structured document-oriented languages to purely data-oriented ones, use an XML syntax. The XML format, however, is verbose and not always ideal for human use. Yet, in many of these new languages, documents are intended to be read and written directly by humans. For this reason, many languages have *two* syntaxes—an XML syntax intended for machine processing and interchange, and an alternative non-XML syntax for human use. This necessitates automated translation in one or both directions.

As a representative example, consider the language RELAX NG [12]. It is a schema language for XML, but we are not interested in the semantics of RELAX NG documents here, only in their syntax. The original language definition specifies an XML syntax, and a later separate specification provides a compact non-XML syntax [11]. A main goal of providing the non-XML syntax is to maximize readability. As an example (taken from the RELAX NG documentation), consider the following tiny RELAX NG document written using the XML syntax:

```
<element name="addressBook"
         xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

In the alternative non-XML syntax, this document looks as follows:

```
element addressBook {
  element card {
    element name { text },
    element email { text }
  }*
}
```

The former can be manipulated by standard XML tools, whereas the latter is more friendly towards human beings. The XML syntax may be formalized by an XML schema language, such as DTD (or RELAX NG itself). The main structure of the non-XML syntax may be formalized using, for example, EBNF.

With the two syntaxes in place, we need to be able to transform documents between them. For RELAX NG, there are numerous implementations of such converters. Converting from the XML syntax to the non-XML syntax, a common approach is to use an XSLT stylesheet. In the other direction, there are no obvious choices, so typically one resorts to programming the conversion in a general purpose programming language, for example Java or Python.

This raises a number of problems: The translations in the two directions are made as two entirely different programs, often even using two different pro-

gramming languages. This requires lots of tedious programming. Also, it makes maintenance difficult in case the syntax evolves. Since the programming languages being used are typically Turing complete (even XSLT is so), it is generally difficult to reason about their correctness. Specifically,

- there is no guarantee that the translations are *reversible* in the sense that translating a document in one direction and then back again will result in the original document (modulo whitespace or similar irrelevant details); and
- there is no guarantee that the translation into the XML syntax always produces documents that are *valid* according to a schema description.

These problems are not specific to the RELAX NG example. Similar situations occur for many other languages, however, RELAX NG is among the more complicated ones.

To attack these problems, we first make an interesting observation. Considering the grammars for the two syntaxes (one given by an XML schema, the other by an EBNF grammar), they commonly have a similar overall structure. The variations mainly occur at the level of individual grammar productions where the two syntaxes may vary in the order of production constituents, choices of literals, and whitespace and other ignorable parts. Notably, there are typically no drastic reorganizations or computations involved when converting one way or the other. In the remainder of this paper, we exploit this in the design of *XSugar*, a system for managing dual syntax of XML languages.

1.1 Contributions

Our contributions are the following.

- We describe the XSugar language and show how it can be used for concisely specifying two-way translations between XML and non-XML syntax.
- We identify conditions for reversibility and present an approach for conservatively checking these conditions.
- Based on previous results on static analysis of XML transformations [7,10,20,19,28], we show that it is possible to statically guarantee validity of output for the translation to XML, using, for example, XML Schema [33,4].
- Using a prototype implementation, we evaluate the approach on a number of real-world examples: RELAX NG, XFlat [34], BibTeXML [16], and Wiki [23].

We imagine various possible usage scenarios of XSugar. Non-XML languages can easily be given an alternative XML syntax for enhancing data interchange; XML-based languages may be given a more human readable non-XML syntax; and, as in the case of RELAX NG, for languages where both syntaxes already exist, XSugar may be used to concisely specify the relation between the two.

Several other projects and technologies are aimed at providing alternative syntax for XML languages. While they have overlapping goals with XSugar, none of them simultaneously consider general two-way translations and static guarantees of validity.

XSLT is often used for translating XML documents into other representations; however, stylesheets are not reversible, so these representations cannot in general be parsed back into XML.

The Presenting XML project [32] provides a domain-specific language for programming transformations between XML and flat files. However, translations are not reversible and, thus, two separate specifications must be maintained for a given dual syntax. The XFlat project [34] has largely the same approach as XSugar, as it allows translations between flat file formats and XML, specified by a single XFlat schema. However, it is restricted to files consisting of sequences of records, rather than general context-free syntax. Section 5.1 contains a more detailed comparison.

The PADS project [24] translates data into other representations, including XML. Data formats are described using a sophisticated calculus that include dependent types and computations—thus going beyond context-free parsing. The translations into XML format are generic, in the sense that the XML schema is generated automatically based on the PADS specification, thus the programmer cannot target an existing XML format. Also, PADS differs from XSugar in that its translations are not automatically reversible.

Other approaches provide bidirectional translations between two XML languages, without considering the case of parsing or generating alternative, non-XML syntax: The biXid project [18] proposes a language inspired by regular expression patterns from XDuce, and the paper [29] presents a framework based on Haskell.

Several projects, such as [13,2,25], suggest an alternative syntax for XML itself, independently of any particular XML language. Such work is only superficially similar to our work, since this alternative syntax is fixed while our is different for each application domain. Program inversion [1] attacks reversibility in a general context, but does not provide a solution to our particular problem.

2 The XSugar Language

We describe the XSugar language by a small example and then explain how to translate between XML- and non-XML syntax based on an XSugar specification.

Assume that we have an XML representation of *student information* as described by the following DTD:

```
<!ELEMENT students (student*)>
<!ELEMENT student (name,email)>
<!ATTLIST student sid CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

All elements belong to the namespace `http://studentsRus.org/`. Additionally, the values of `name`, `email`, and `sid` are required to satisfy some extra syntactic requirements, which we describe later. The syntax can also be formalized using XML Schema as shown in Section 4. The following is an example of a valid document:

```
<students xmlns="http://studentsRus.org/">
  <student sid="19701234">
    <name>John Doe</name>
    <email>john_doe@notmail.org</email>
  </student>
  <student sid="19785678">
    <name>Jane Dow</name>
    <email>dow@bmail.org</email>
  </student>
</students>
```

There is also an alternative non-XML syntax for this document:

```
John Doe (john_doe@notmail.org) 19701234
Jane Dow (dow@bmail.org) 19785678
```

Here, each student corresponds to one line. The name is written first, then the email address in parentheses surrounded by whitespace, and finally the ID. Notice that the ordering of the constituents differs from the XML version.

With XSugar, we can concisely specify the connection between the two syntaxes:

```
xmlns = "http://studentsRus.org/"

Name   = [a-zA-Z]+(\ [a-zA-Z]+)*
Email  = [a-zA-Z._]+\@[a-zA-Z._]+
Id     = [0-9]{8}
NL     = \r\n|\r|\n

file : [persons p] = <students> [persons p] </>
```

```
persons : [person p] [NL] [persons more] =
          [person p] [persons more]
        : =

person : [Name name] _ "(" [Email email] ")" _ [Id id] =
         <student sid=[Id id]>
           <name> [Name name] </>
           <email> [Email email] </>
         </>
```

The first line declares the namespace associated with the empty prefix. The next four lines define some *regular expressions*, which are used for describing syntactic tokens. For example, `Name` matches one or more blocks of alphabetic characters, separated by space characters. The remaining lines define *grammar productions*, each having the form

$$\textit{nonterminal} \ : \ \alpha \ = \ \beta \ ;$$

(If the nonterminal is omitted in a production, the one from the preceding production is assumed.) Unlike ordinary grammar productions, we here have *two* right-hand sides—one for the non-XML syntax (α) and one for the XML syntax (β).

The α part is generally a sequence of *items* of the form `[`X *name*`]` or `[`X`]`, where X is either a nonterminal or a regular expression name, and of quoted literals such as `"("` and `")"` above. Additionally, the special character `_` is used for describing whitespace, which we return to later. The item names (for example, `email` above) are used for connecting the non-XML and the XML descriptions, as explained below.

The β part consists of an *XML template*, which is a fragment of well-formed XML that may contain items in place of attribute values (like `sid=[Id id]` in the example) and in element content (like `[Email email]`). End tags are written `</>` for brevity. We also allow dynamic element and attribute names (written as `[`X *name*`]` in place of the name).

The nonterminal or regular expression name associated with a given item name must be the same in both α and β. We use the convention that regular expression names start with a capital letter (such as `Email`), and nonterminals start with a lower case letter (such as `persons`). Special characters can be escaped with a backslash notation, with Unicode character numbers, or with XML-style character references.

Notice that if we ignore the β part in every production and the *name* part in every item, an XSugar specification $\mathcal{S}$ is essentially an ordinary BNF-like context-free grammar $\mathcal{S}_\alpha$ (where the first occurring nonterminal is the start nonterminal). This grammar specifies the non-XML syntax of the language.

Conversely, we obtain a grammar $\mathcal{S}_\beta$ for the XML syntax by ignoring the α parts. Literals and unnamed items correspond to information that has no counterpart in the opposite grammar. For both grammars, we require all nonterminals to be productive. For later use, we assume that the productions in $\mathcal{S}$ are implicitly indexed in order of occurrence.

As an extension of the notion of grammars presented above, we also allow *unordered* productions: In a production where the delimiter `:&` appears in place of `:`, the α part is unordered, meaning that it matches any permutation of the constituents. Similarly, if the `=` symbol is replaced by `=&`, the β part is unordered. We show a use of unordered productions in Section 5.3.

Also, productions may be given *priorities* to allow disambiguation during parsing. If the `:` symbol in a production is replaced by the symbol `>:` then it is given lower priority than all previously occurring productions for the same nonterminal. This feature is used in Section 5.4.

Figure 1 shows the abstract syntax for XSugar specifications.

2.2 Transforming via Unifying Syntax Trees

An XSugar specification $\mathcal{S}$ defines a translation from the non-XML syntax to the XML syntax and vice versa. This translation goes via a *unifying syntax tree* (UST), which abstracts away the ordering of the constituents of each grammar production and also ignores parts corresponding to literals and unnamed items. More precisely, a UST is an unordered labeled tree of nodes where each node is either a *terminal node* or a *nonterminal node*. A terminal node is a leaf that is labeled with a string. A nonterminal node is labeled with a nonterminal, each edge to a child node is labeled with an item name, and every node has at most one outgoing edge with a given item name. Moreover, every nonterminal node is labeled with a production index, which we will need later.

Assume that we want to transform a text x from the non-XML syntax to the XML syntax. This is done in two steps:

- first *parse* the text x according to $\mathcal{S}_\alpha$, yielding a UST u;
- then *unparse* u relative to $\mathcal{S}_\beta$ yielding the resulting XML document.

The other direction—translating from XML syntax to non-XML syntax—is symmetric. The processes of parsing and unparsing with USTs is illustrated in Figure 2 and described in the following sections. As an example of a UST, the one corresponding to the example student information document is shown in Figure 3.

```
Grammar        →  NamespaceDecl* RegexpDef* Production*
NamespaceDecl  →  xmlns PREFIX? = " URI "
RegexpDef      →  REGEXP = Regexp
Production     →  NONTERMINAL? >? : &? LeftItem* = &? RightItem*
LeftItem       →  Item
RightItem      →  Item
               |  ElementItem
Item           →  " STRING "
               |  [ NONTERMINAL NameOrExample? ]
               |  [ REGEXP NameOrExample? ]
               |  _
               |  __
ElementItem    →  < Name AttributeItem* > RightItem* </>
               |  < Name AttributeItem* />
AttributeItem  →  Name = " STRING "
               |  Name = [ NONTERMINAL NameOrExample? ]
               |  Name = [ REGEXP NameOrExample? ]
Name           →  NCNAME
               |  PREFIX : NCNAME
               |  [ REGEXP NameOrExample? ]
NameOrExample  →  ITEM
               |  " STRING "
```

Fig. 1. Abstract syntax for XSugar specifications. We here use * and ? to denote "zero-or-more" and "optional", respectively. The syntax of regular expressions (*Regexp*) is as in the BRICS Automaton package [31]. The nonterminals *PREFIX* and *NCNAME* describe valid prefixes of qualified XML names and local names, respectively); *REGEXP*, *NONTERMINAL*, and *ITEM* are names of regular expressions, nonterminals, and items, respectively; *STRING* consists of literal strings; and _ and __ describe optional whitespace and nonempty whitespace, respectively.

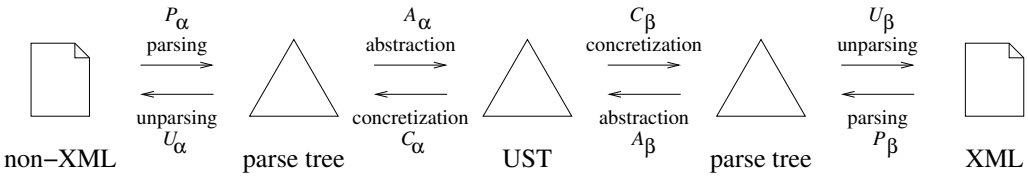

Fig. 2. The transformation process.

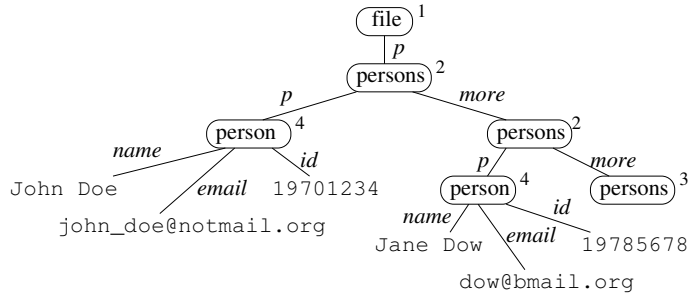

Fig. 3. UST for the student information document.

Parsing

Given a text x and a grammar $\mathcal{S}_i$ (where i is either α or β, depending on which direction we are translating), we construct the UST u as follows. First, we run a context-free-grammar parser on x and $\mathcal{S}_i$, yielding an ordinary parse tree t. Since we work with the full class of context-free languages and do not apply lexical analysis, we use Earley-style parsing [14] (rather than, for example, LALR(1) parsing). The parser must of course be equipped to handle our extensions with unordered productions and priorities. If $\mathcal{S}_i$ is ambiguous, t is chosen arbitrarily among the possibilities but respecting priorities; we discuss ambiguity further in Section 3.

Note that we use the same parsing technique for both directions. Using context-free grammar parsing for XML documents allows us to describe, in a uniform framework, structure of character data and attribute values, which would not be possible with a conventional XML parser.

From the parse tree, we construct the UST u as follows.

- Every parse tree node corresponding to a named regular expression item in $\mathcal{S}_i$ becomes a terminal node labeled with the corresponding string.
- Every parse tree node corresponding to a named nonterminal item in $\mathcal{S}_i$ becomes a nonterminal node. Its label is the nonterminal, and its index is the index of the associated grammar production of the parse tree node. For each named item in the production, a child edge with that name is made to the UST node of the corresponding child node in the parse tree.

All parse tree nodes corresponding to literals or unnamed items are ignored in the construction. Figure 4 shows the parse tree for the student example which yields the UST shown in Figure 3. Note that the parse tree as opposed to the UST is ordered and includes values of ignorable items.

The whitespace marker `_` is implicitly defined as an abbreviation of the unnamed regular expression item `[OPT_WHITESPACE]` where `OPT_WHITESPACE` is the regular expression `[ \t\r\n]*` (that is, strings of whitespace charac-

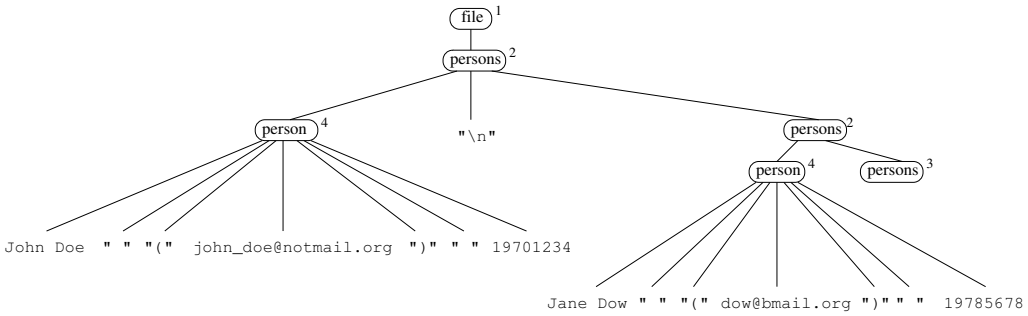

Fig. 4. Parse tree for the non-XML student information document.

ters). Similarly, `__` refers to `WHITESPACE`, which represents *nonempty* strings of whitespace. For convenience, some other widely used regular languages are also built in: `NCNAME`, `QNAME`, `CHAR`, `NAMECHAR`, `LETTER`, and `URI` correspond to central syntactic categories found in the specifications of XML [9] and XML Namespaces [8].

In case x is an XML document and $i = \beta$, we initially *normalize* both x and $\mathcal{S}_\beta$ in a process that resembles XML canonicalization [5] but results in a different representation:

(1) whitespace inside tags (but outside attribute values) is reduced to a minimum;
(2) attribute values are enclosed by double quotes (that is, `sid='19701234'` is changed to `sid="19701234"`);
(3) the short form of empty elements is expanded (for instance, `<p/>` becomes `<p></p>`);
(4) character encoding is set to UTF-8;
(5) character and entity references are expanded;
(6) all whitespace character data is removed in elements that do not contain non-whitespace character data;
(7) XML comments, XML declarations, processing instructions, DOCTYPEs are removed, and end tag names are shortened to `</>`; and
(8) all qualified names are expanded to the form $\{URI\}localname$.

This normalization allows us to disregard the many equivalent forms that XML documents may have. (Attribute order is handled using the unordered production mechanism since we allow non-constant attribute names.) As an example, the following XML document is equivalent to the one shown earlier and is also parsed by the student example grammar:

```
<stu:students xmlns:stu="http://studentsRus.org/">
  <!-- this is not normalized -->
  <stu:student     sid='19701234' >
        <stu:name>John Doe</stu:name>
        <stu:email>john_doe&#64;notmail.org</stu:email>
  </stu:student>
```

```
    <stu:student sid='19785678'>
      <stu:name>Jane Dow</stu:name>
      <stu:email>dow@bmail.org</stu:email>
    </stu:student>
  </stu:students>
```

Focusing on just the first `student` element and its contents, normalization will rewrite it as follows (without the line breaks):

```
<{http://studentsRus.org/}student sid="19701234">
<{http://studentsRus.org/}name>John Doe</>
<{http://studentsRus.org/}email>john_doe@notmail.org</></>
```

Unparsing

Given a UST u and an XSugar specification $\mathcal{S}$ where u has been generated from either $\mathcal{S}_\alpha$ or $\mathcal{S}_\beta$, we construct an ordinary parse tree t as a concretization of u relative to $\mathcal{S}_i$ as follows, starting at the root of u.

- A terminal node in u becomes a parse tree leaf node labeled with the same string.
- A nonterminal node with index k becomes a parse tree node labeled with the same nonterminal and index. For each component in the production with index k in $\mathcal{S}_i$ in order, a corresponding subtree is constructed depending on the component kind:
  - for a named item, the subtree is constructed recursively from the child UST node with that name;
  - for an unnamed regular expression item, the subtree is a leaf node labeled with an arbitrary string matching the regular expression (for example, a shortest one);
  - for an unnamed nonterminal item, the subtree is chosen as an arbitrary parse tree derivable from the corresponding nonterminal in $\mathcal{S}_i$; and
  - for a literal, the subtree is a leaf node labeled with the literal string.

Notice that unnamed items are handled by generating arbitrary representatives. This makes sense since such items describe information that only occurs in one of the two syntaxes. To add more control to the unparsing process, XSugar also allows unnamed items of the form `[`X `"`s`"]`, where s is a string that belongs to the language defined by X. The unparser will then simply select s as a representative for X. Continuing the student example, the UST from Figure 3 is unparsed into the XML parse tree show in Figure 5.

Once we have the parse tree t, the resulting text x is simply the concatenation of the text in the leaves. A few technical issues remain: We escape or unescape (depending on the direction of translation) special XML characters to ensure that, for example, the character `<` in non-XML corresponds to `<` in XML. Also, tag names are inserted in the end tags.

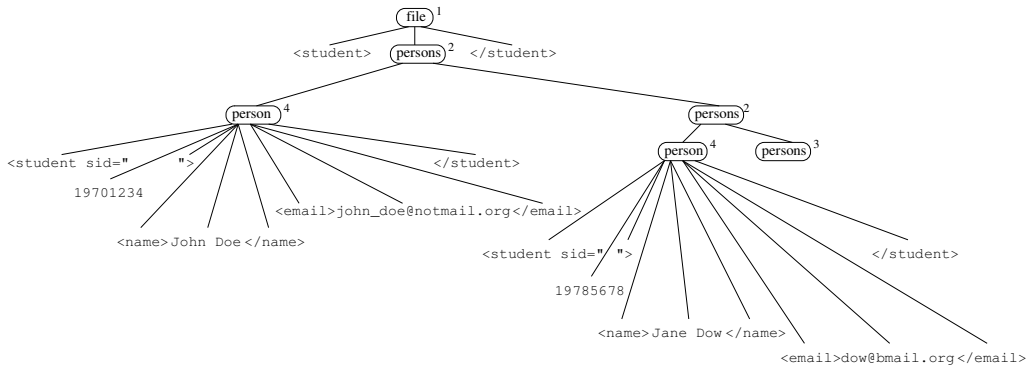

Fig. 5. XML parse tree for the student information document.

3 Reversibility

A desirable property of an XSugar specification is that it is *reversible*, meaning that no information is lost when transforming data in either direction. Below, we formally define a notion of reversibility and describe a static analysis for conservatively checking this property on XSugar specifications.

One way to define reversibility is that performing a roundtrip from one syntax to the other and back should always yield the exact same document. Using the names of the individual transformation steps shown in Figure 2, define

$$T_{\alpha \rightarrowtail \beta} = U_\beta \circ C_\beta \circ A_\alpha \circ P_\alpha$$

$$T_{\alpha \leftarrowtail \beta} = U_\alpha \circ C_\alpha \circ A_\beta \circ P_\beta$$

Reversibility then means that both $T_{\alpha \rightarrowtail \beta} \circ T_{\alpha \leftarrowtail \beta}$ and $T_{\alpha \leftarrowtail \beta} \circ T_{\alpha \rightarrowtail \beta}$ are identity transformations (on non-XML data and XML data, respectively). This condition is equivalent to $T_{\alpha \rightarrowtail \beta}$ being a bijection. Verifying this property is equivalent to ensuring that the four individual steps P_α (parsing), A_α (abstraction), C_β (concretization), and U_β (unparsing) are all bijections.

However, as explained in Section 2.2, USTs explicitly discard three kinds of information from the input documents:

(1) information removed by normalization of XML documents;
(2) information that corresponds to unnamed items; and
(3) information represented by the order of constituents described by unordered productions.

For this reason we need to loosen the previous definition of reversibility as follows: Performing a roundtrip from one syntax to the other and back should always yield the exact same document, *modulo loss of the above information*. Define that two input documents, x_1 and x_2, (both XML or both non-XML)

are *equivalent* if their USTs are identical, that is $A_i \circ P_i(x_1) = A_i \circ P_i(x_2)$. Also, define two parse trees, t_1 and t_2, to be equivalent if their USTs are identical, that is $A_i(t_1) = A_i(t_2)$. Reversibility now corresponds to the four individual steps P_α, A_α, C_β, and U_β being bijections on the induced equivalence classes, which can be checked as follows:

Checking bijectivity of A_i and C_i: Since all UST tree nodes are explicitly annotated with production indices and all edges to subtrees are labeled with item names, we simply have to check that, for each production, all item names are used exactly once on the other side (ignoring unnamed items).

Checking bijectivity of P_i and U_i: This corresponds to checking that the context-free grammar $\mathcal{S}_i$ is unambiguous: P_i is by definition injective (a parse tree retains all information from the input, except for XML normalization), and U_i is injective if and only if $\mathcal{S}_i$ is unambiguous (since there exists a string with two nonequivalent parse trees exactly when unparsing is not injective).

The ambiguity decision problem for context-free grammars is undecidable [17]. We rely on a static analysis to conservatively approximate the problem, as explained in Section 3.1. Aside from the reversibility issue, static detection of ambiguity is useful by itself since ambiguity in XSugar grammars is rarely intended.

An interesting consequence of performing a roundtrip transformation is that it results in a canonical representative for the equivalence class of the input document, as defined by the concretization and unparsing steps.

Note that if the reversibility condition does not hold, the XSugar transformation still runs but with the following consequences: if the abstraction step is not bijective, then information may be lost in the translation; if the parsing step is not bijective, then the parser will just pick one of the possible parse trees.

3.1 Ambiguity Analysis

As explained above, the only remaining challenge in checking reversibility is to devise a useful approximation technique for deciding unambiguity of a given context-free grammar. We prefer an approximation that is conservative in the sense that when it reports a grammar unambiguous, then it is truly so.

A classical algorithm is the LR(k) check [22]. However, to obtain a simple and expressive language definition, XSugar employs scannerless parsing where grammars are expressed on individual characters rather than tokens. In this situation, LR(k) and its variants are inadequate since they rely on a fixed

lookahead and hence cannot distinguish between, for example, two productions that both begin with an identifier. Additionally, the unordered productions and production priority mechanisms in XSugar are not easily incorporated into the LR(k) method.

For these reasons, we use an alternative approach, described in the paper [6] and briefly summarized in the following. Ambiguity of context-free grammars may be fully characterized by two (also undecidable) predicates: *vertical unambiguity* and *horizontal unambiguity*. Vertical unambiguity means that no two productions of any nonterminal can derive the same string. Horizontal unambiguity means that no production right-hand-side p may be split in two nonempty parts $p_L p_R = p$ such that there exists a string xay, where a is nonempty, that may be ambiguously parsed by having either p_L derive xa and p_R derive y or by having p_L derive x and p_R derive ay. This gives a linguistic characterization of ambiguity in the sense that we can now consider the ambiguity problem in terms of relationships between languages at various points in the grammar. In particular, it allows the use of approximations of context-free grammars [26], which sacrifices completeness but gives decidability.

The complications we mentioned earlier that precluded usage of the LR(k) algorithm are all easily dealt with in this approach. First, it works smoothly with scannerless parsing. Second, for the unordered productions we simply test horizontal ambiguity locally for all combinations for the production. Third, the production priority feature is easily handled by simply omitting the corresponding vertical ambiguity checks.

3.2 Examples

Running the reversibility analysis on our student example from Section 2 produces the following encouraging output:

```
Transformation is guaranteed to be reversible!
```

As an illustration of the error messages that our approach is able to give, we make some erroneous variations of the student example.

First, if we remove the `sid=[Id id]` attribute in the `student` element, the abstraction step will become non-bijective. This causes the following error message to appear:

```
*** Reversibility error
Source: students.xsg line 14 column 52
Error: information loss from non-XML to XML:
 item named id missing in XML grammar
```

Next, let us introduce grammar ambiguity by adding another production for the `file` nonterminal having a dot describe the "empty student record" and forgetting that the `persons` nonterminal already featured an empty case:

```
file : [persons p] = <students> [persons p] </>
     : "."         = <students> </>
```

Running the analysis on the XML grammar yields the following ambiguity report:

```
*** Reversibility error
Source: students.xsg line 8 column 1 and
 students.xsg line 9 column 1 and
Error: vertical ambiguity in XML grammar:
 XML string <{http://studentsRus.org/}students></>
 corresponds to non-XML strings "" and "."
```

The XML string being shown clearly has two parses corresponding to the two different `file` productions which, in turn, means that the syntactic alternative is non-reversible.

Additionally, we might introduce another error by using the following definition of a student record where the alternative syntax is condensed by omitting parentheses and forgetting whitespace:

```
person : [Name name] [Email email] [Id id] =
         <student sid=[Id id]>
           <name> [Name name] </>
           <email> [Email email] </>
         </>
```

Running the analysis now yields this report:

```
*** Reversibility error
Source: students.xsg line 14 column 22
Error: horizontal ambiguity in non-XML grammar:
 non-XML string "AAA@A.A00000000"
 corresponds to XML strings
 <{http://studentsRus.org/}student sid="00000000">
 <{http://studentsRus.org/}name>A</>
 <{http://studentsRus.org/}email>AA@A.A</></>
 and
 <{http://studentsRus.org/}student sid="00000000">
 <{http://studentsRus.org/}name>AA</>
 <{http://studentsRus.org/}email>A@A.A</></>
```

The example string describes either a person `A` with email `AA@A.A` or a person `AA` with email `A@A.A`. Clearly, such messages are useful for detecting and eliminating errors in the translations.

We have run the ambiguity analysis on our other examples where it uncovered an interesting error in our description of the alternative syntax for RELAX NG:

```
*** Reversibility error
Source: relax.xsg line 103 column 1 and
 relax.xsg line 104 column 1
Error: vertical ambiguity in non-XML grammar:
 non-XML string "A"
 corresponds to XML strings
 <{http://relaxng.org/ns/structure/1.0}name>A</>
 and
 <{http://relaxng.org/ns/structure/1.0}name>A</>
```

By inspecting the mentioned lines in `relax.xsg`, this error message tells us that the (non-XML) string `A` could be interpreted either as an `Identifier` or as an `NCNAME` (although they, in this case, happen to have the same XML syntax). This ambiguity was then fixed by introducing priority for one of the productions.

Since the ambiguity check is approximate, false positives are possible. We have encountered this in the RELAX NG example where the analysis reports 11 potential ambiguity problems even though the grammar is in fact unambiguous. The approach presented in [6] allows precision to be improved by manually specifying grammar unfolding transformations, which for the RELAX NG example cause all false positives to disappear.

4 Static Validation

Assume that an XML language, described by some schema formalism, has been given an alternative non-XML syntax using XSugar. An obvious *validation* requirement is that the translations of non-XML documents must always result in valid XML data, relative to the schema. The XSugar tool can perform this check by analyzing the XML syntax. The analysis is exact: it reports success if and only if syntactically correct non-XML input (according to $\mathcal{S}_\alpha$) always results in valid XML output (according to the XML schema).

Our static analysis is based on previous results [7,10,20,19,28] where the concept of *XML graphs* (also called *summary graphs* when used in program analysis) is used to model sets of XML documents. Intuitively, an XML graph is reminiscent of an XML tree but may contain loops and choices. Also, element and attribute names, attribute values, and character data are described by regular string languages. We have an algorithm [21] that is able to check that every document described by an XML graph is valid according to a schema written in DTD or XML Schema. (An earlier version was based on DSD2 schemas instead [27].)

More precisely, an XML graph consists of nodes of various kinds:

element: roughly corresponds to Element nodes in XML DOM [3], labeled with a regular string language describing the element name and having a child node describing attributes and contents;
attribute: resembles Attribute nodes in XML DOM, but, as for element nodes, attribute names are described with regular string languages;
sequence: describes an ordered sequence of nodes;
text: as Text nodes in XML DOM, but labeled with a regular string language rather than a single string;
interleave: as sequence but for unordered sequences;
choice: describes a union of the sets of XML documents being described by the children.

(The actual definition also contains some other node kinds and information that we do not need here; for details, see [28].)

From an XSugar specification (see Figure 1), it is simple to extract an XML graph that precisely represents all XML documents that can be generated by the $\mathcal{S}_\beta$ grammar:

- each nonterminal becomes a choice node with a child for each of its productions;
- a production becomes a sequence node if ordered and an interleave node if unordered, and a child node is made for each item;
- for a nonterminal item `[`X ...`]`, the node is the one corresponding to the nonterminal X;
- for a regular expression item `[`X ...`]`, the node is a text node labeled with the regular expression of X, and quoted literal items and whitespace items are treated as regular expression items;
- for an element item, the node is an element node with a corresponding name and with a `sequence` child node describing the attributes and contents, and attributes similarly become attribute nodes.

With this translation, the language of the resulting XML graph (as defined in [28]) is equal to the language of $\mathcal{S}_\beta$. As a simple optimization, we may omit choice nodes and sequence nodes that have exactly one child. For the student information example from Section 2.1, the grammar $\mathcal{S}_\beta$ for the XML syntax is

```
file : <students> [persons p] </>

persons : [person p] [persons more]
        :
```

```
person : <student sid=[Id id]>
           <name> [Name name] </>
           <email> [Email email] </>
         </>
```

and the resulting XML graph looks as shown in Figure 6.

Static validation is performed by checking the XML graph against the given XML schema, which may be written in XML Schema (rather than using the less precise DTD version shown in Section 2.1):

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://studentsRus.org/"
        xmlns:s="http://studentsRus.org/"
        elementFormDefault="qualified">

  <element name="students">
    <complexType>
      <sequence minOccurs="0" maxOccurs="unbounded">
        <element ref="s:student"/>
      </sequence>
    </complexType>
  </element>

  <element name="student">
    <complexType>
      <sequence>
        <element name="name" type="s:Name"/>
        <element name="email" type="s:Email"/>
      </sequence>
      <attribute name="sid" type="s:Id"/>
    </complexType>
  </element>
```

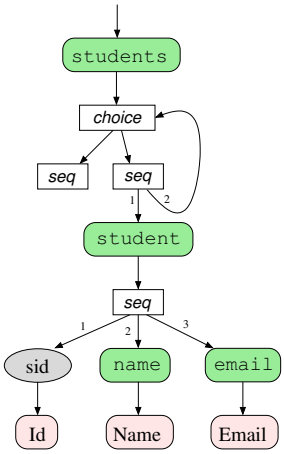

Fig. 6. XML graph for the student information example.

```
  <simpleType name="Id">
    <restriction base="string">
      <pattern value="[0-9]{8}"/>
    </restriction>
  </simpleType>

  <simpleType name="Name">
    <restriction base="string">
      <pattern value="[a-zA-Z]+( [a-zA-Z]+)*"/>
    </restriction>
  </simpleType>

  <simpleType name="Email">
    <restriction base="string">
      <pattern value="[a-zA-Z._]+@[a-zA-Z._]+"/>
    </restriction>
  </simpleType>
</schema>
```

On this example, the validator produces this result:

```
XML output is guaranteed to be valid!
```

If we had made some mistakes, for example changed the definition of `Id` to `[0-9]{5,8}` and swapped the order of the `name` and `email` elements in the XSugar specification, the output would instead be like this:

```
*** Validation error
Source: element {http://studentsRus.org/}student at
 students.xsg line 15 column 10
Schema: students.xsd line 20 column 7
Error: invalid attribute value: sid="00000"

*** Validation error
Source: element {http://studentsRus.org/}student at
 students.xsg line 15 column 10
Schema: students.rng line 16 column 7
Error: invalid contents:
 <{http://studentsRus.org/}email/><{http://studentsRus.org/}name/>
```

Clearly, such error messages are useful for locating and correcting the errors. Another example of validation is discussed in Section 5.3.

The dual validation check—that output in the XML to non-XML direction is always syntactically correct—only makes sense if the alternative syntax is already described by a different context-free grammar. As shown in Section 5.2, this is the case for RELAX NG, where the original grammar must be rewritten slightly to allow the XSugar translation. However, the inclusion test between

context-free grammars is undecidable [17], and we are not aware of useful approximation algorithms.

We may also consider *coverage* checks, which for the XML to non-XML direction means that every XML document described by the external schema can be parsed by the XSugar grammar. However, this generally requires checking inclusion of a regular language in a context-free language, which is also undecidable [17]. The dual coverage check is just the opposite inclusion check between the two context-free grammars. Note that in many situations the alternative syntax is *defined* by the XSugar specification, in which case both non-XML to XML coverage and XML to non-XML validation come for free.

5 Evaluation

We have implemented a fully functional prototype of the XSugar tool. It is available for download—along with all the examples presented in this paper—at the XSugar Web site:

`http://www.brics.dk/xsugar/`

The underlying parser [30] is a variation of Earley's algorithm [14] that builds a UST directly without the intermediate ordinary parse tree, has explicit support for regular expression items, unordered productions, and production priorities. The tool also performs the static validation described in Section 4 by means of the validation system developed for the XACT system [21,19], and it checks grammar ambiguity using the technique described in the paper [6].

In the following sections, we present a range of examples showing how XSugar may be used for concrete XML languages. Each example highlights certain features of the XSugar tool. The complete XSugar specifications are available at the URL mentioned above, along with examples of input documents and schemas.

5.1 XFlat

The XFlat system [34] allows translations between flat file formats and XML, specified by a single XFlat schema. As an example, the translation between these two formats

```
123456789,"Doe, John",100000.00
444556666,"Average, Joe",53000.00
```

```
<employees>
  <employee>
    <ssn>123456789</ssn>
    <name>Doe, John</name>
    <salary>100000.00</salary>
  </employee>
  <employee>
    <ssn>444556666</ssn>
    <name>Average, Joe</name>
    <salary>53000.00</salary>
  </employee>
</employees>
```

is specified by the following XFlat schema:

```
<XFlat Name="employees_schema" Description="Schema for CSV flat file">
  <SequenceDef Name="employees" Description="employees flat file">
    <RecordDef Name="employee" FieldSep="," RecSep="\N" MaxOccur="0">
      <FieldDef Name="ssn" NullAllowed="No"
                MinFieldLength="9" MaxFieldLength="11"
                DataType="Integer" MinValue="0" QuotedValue="Yes"/>
      <FieldDef Name="name" NullAllowed="No" QuotedValue="Yes"/>
      <FieldDef Name="salary" NullAllowed="No"
                DataType="Float" MinValue="0" QuotedValue="Yes"/>
    </RecordDef>
  </SequenceDef>
</XFlat>
```

Each such schema may systematically be translated into an equivalent XSugar description, which for the above example looks as follows:

```
SSN    = [0-9]{9,11}
Name1  = [^",]*
Name2  = [^"]*
Salary = [0-9]+("."[0-9]+)?

file : [employees es] = <employees> [employees es] </>

employees : [employee e] [employees es] =
            [employee e] [employees es]
          : =

employee : [SSN x] "," [name y] "," [Salary z] "\n" =
           <employee>
             <ssn> [SSN x] </>
             <name> [name y] </>
             <salary> [Salary z] </>
           </>
```

```
name  : [Name1 y] = [Name1 y]
     >: "\"" [Name2 y] "\"" = [Name2 y]
```

The XSugar version differs from the XFlat version in one respect. The XFlat translation from XML to flat file format is ambiguous, since quotes around fields are optional, unless the field value contains a comma. In our version, quotes are only added when they are necessary, thanks to the specification of priorities between the two `name` productions.

In other respects, the XSugar tool is more general than XFlat. First, it may handle context-free syntax. Second, even in the niche of flat files, it may perform more general translations. For example, an XSugar translator could parse up the first and last names and swap their order within the field, which is not possible using XFlat.

5.2 RELAX NG

As mentioned in the introduction, the RELAX NG schema language has an alternative syntax [11], which may be expressed by an XSugar specification. The α-grammar is relatively close to the one given in the RELAX NG specification, but some massaging was required to accommodate the local translations that XSugar supports. For example, the official EBNF for the compact syntax contains the following productions:

```
pattern ::=   ...
            | pattern ("," pattern)+
            | pattern ("&" pattern)+
            | pattern ("|" pattern)+
            | pattern "?"
            | pattern "*"
            | pattern "+"
```

In the translation, maximal nonempty sequences of patterns separated by `,` must be enclosed by `<group>` tags, those separated by `&` by `<interleave>` tags, and those separated by `|` by `<choice>` tags. Furthermore, the three operators must satisfy an operator precedence hierarchy. This translation is only possible in XSugar if the grammar is made more explicit in the following manner:

```
pattern ::=        cpattern
cpattern ::=       gpattern "|" crestpattern
                 | gpattern
crestpattern ::=   gpattern "|" crestpattern
                 | gpattern
gpattern ::=       ipattern "," grestpattern
                 | ipattern
```

```
grestpattern ::=   ipattern "," grestpattern
                 | ipattern
ipattern ::=       upattern "&" irestpattern
                 | upattern
irestpattern ::=   upattern "&" irestpattern
                 | upattern
upattern ::=       bpattern
                 | bpattern "?"
                 | bpattern "*"
                 | bpattern "+"
bpattern ::=       ...
```

Here, the operator precedences and associativities are expressed in the usual manner by introducing extra nonterminals, and the grammar is further unfolded to allow us to distinguish between the first and the rest of maximal sequences. Operator precedences may in simple cases be captured by production priorities, but general expression languages will typically require some grammar unfolding. This particular example requires by far the most complex unfoldings that we have yet encountered.

On the RELAX NG site, a translation from compact to ordinary syntax is defined by an XSLT stylesheet of 894 lines. The inverse translation is defined by a Python script of 1,478 lines. In all, that implementation stacks up to 2,372 lines of code, while the XSugar description is only 123 lines (a factor of 1:19). On top of this succinctness, the XSugar solution is easier to maintain and delivers all the safety guarantees discussed in Sections 3 and 4.

5.3 BibTeXML

The BibTeXML project [16] provides an XML-syntax for the popular BibTeX bibliography format. The XML format is quite complex and is described in 400 lines of DTD notation. This dual syntax is also a larger example of an XSugar specification, totaling 750 lines.

The example is noticeable in two respects. First, it involves some fairly intricate parsing and translation. For example, a list of authors may be separated by the word `and`, and first and last names may be written either directly or in reverse order separated by commas. Each component of a name is built from several individual parts, each of which is a string that does not contain special BibTeX characters or is equal to the word `and`. A part may optionally be enclosed in brackets, and the special character `~` is used to denote an explicit space character. In the translation to XML, each author must be enclosed by a separate `author` element and the names must be normalized. This is obtained by the following dual syntax:

```
AND = [Aa][Nn][Dd]
PART = ([^",{}&<>~ \n\t]+)&~<AND>

authors : [name n] = <bibxml:author> [name n] </>
        : [name n] [AND] [authors as] =
          <bibxml:author> [name n] </> [authors as]

name : [parts ps] = [parts ps]
     : [parts last] _ "," _ [parts first] =
       [parts first] __ [parts last]

parts : [PART p] = [PART p]
      : "{" [PART p] "}" = [PART p]
      : "~" = " "
      : "\n" = " "
      : [PART p] [parts ps] = [PART p] [parts ps]
      : _ =
```

Second, a BibTeX file allows an arbitrary mix of fields, whereas the XML version requires (for some reason) a specific order. This is a situation where the unordered productions are useful:

```
ARTICLE = [Aa][Rr][Tt][Ii][Cc][Ll][Ee]
ID = [^ \n\t]+

article : "@" [ARTICLE] _ "{" _ [ID id] _ ","
          _ [articlefields fs] _ "}" =
          <bibxml:entry id=[ID id]>
            <bibxml:article>
              [articlefields fs]
            </>
          </>

articlefields :& [author author] [title title] [journal journal]
                 [year year] [volume volume] ... =
                 [author author] [title title] [journal journal]
                 [year year] [volume volume] ...
```

Note that only the non-XML production is unordered in this case.

In both these situations, the BibTeX format is more liberal than the BibTeXML format. Thus, the translation from BibTeXML to BibTeX will automatically choose a canonicalized representation.

Static validation of the generated XML documents is for this substantial example performed in a few seconds on a standard PC. The analysis discovered four true errors in the definition of the BibTeX translation (despite our best efforts at defining it correctly), which were subsequently corrected.

The Wiki notation [23] is an alternative syntax for Web pages that is used to simplify online collaborative editing. An example is the following description of XSugar (where we have used the Wikipedia dialect):

```
== XSugar ==
The [http://www.brics.dk/xsugar XSugar] project has developed a
notation for specifying a ''dual syntax'' for an [[XML|XML]] language.

An XSugar specification gives rise to the following tools:
* a translation from XML to non-XML syntax
* a translation from non-XML to XML syntax
* a check that these translations are reversible
* static validation of the generated XML documents
```

Part of the functionality of a Wiki tool is to translate such notation into XHTML which is then published:

```
<html>
  <head><title>Wiki</title></head>
  <body>
    <h1>XSugar</h1>
    The <a href="http://www.brics.dk/xsugar">XSugar</a> project has
    developed a notation for specifying a <i>dual syntax</i> for an
    <a href="XML">XML</a> language.<p/>
    An XSugar specification gives rise to the following tools:
    <ul>
      <li>a translation from XML to non-XML syntax</li>
      <li>a translation from non-XML to XML syntax</li>
      <li>a check that these translations are reversible</li>
      <li>static validation of the generated XML documents</li>
    </ul>
  </body>
</html>
```

The essence of such a Wiki tool can be expressed through an XSugar specification; however, certain specialized features must be handled separately, such as the interpretation of user preferences and the conversion of LaTeX fragments into inlined images.

The Wiki notation has an intricate syntax in which newlines and whitespace are significant. To handle these aspects in an unambiguous grammar, it has proved invaluable to use the production priority mechanism. For example, part of the XSugar specification for Wiki looks as follows:

```
flat >: "''''''" _ [words ws] _ "''''''" = <b> <i> [words ws] </> </>
     >: "''''" _ [words ws] _ "''''" = <b> [words ws] </>
     >: "''" _ [words ws] _ "''" = <i> [words ws] </>
     >: "<tt>" _ [words ws] _ "</tt>" = <tt> [words ws] </>
     >: "<sup>" _ [words ws] _ "</sup>" = <sup> [words ws] </>
     >: "<sub>" _ [words ws] _ "</sub>" = <sub> [words ws] </>
     >: "<big>" _ [words ws] _ "</big>" = <big> [words ws] </>
     >: "<br>" = <br/>
     >: [SPACE] = __
     >: "[" [URL u] __ [words ws] "]" =
        <a href=[URL u]> [words ws] </>
     >: "[[Image:" [WORD w] "|" [words ws] "]]" =
        <img src=[WORD w] alt=[words ws]/>
     >: "[[" [WORD w] "|" [words ws] "]]" =
        <a href=[WORD w]> [words ws] </>
     >: [WORD w] = [WORD w]
```

Several of these right-hand sides overlap or are even included in each other, but the grammar is easy to construct since our notion of production priorities coincides with our intuitions as grammar authors.

As mentioned, the translation from Wiki notation to XHTML does not provide the full functionality of a Wiki tool, but an implementation could be built around the XSugar specification. However, in this case the reverse translation from XHTML to Wiki notation is actually more interesting, since it has the effect of performing a maximal *wikification*. This means that XHTML documents are translated into Wiki notation as far as possible, which could be useful when importing external documents into a Wiki context. Again, the priority mechanism is crucial in ensuring that the relevant XHTML tags are not just carried through to the text version.

6 UST Transformations

Our experiences with XSugar suggest that non-local UST transformations may extend its expressive power. Concretely, we have looked at functions θ on USTs that satisfy the following restrictions:

$$\forall t : \theta(A_\alpha(P_\alpha(t))) = \theta^2(A_\alpha(P_\alpha(t))) \qquad \text{(non-XML idempotency)}$$

$$\forall x : \theta(A_\beta P_\beta(x))) = \theta^2(A_\beta(P_\beta(x))) \qquad \text{(XML idempotency)}$$

Such a function induces an equivalence relation of USTs and may thus be incorporated into the reversibility framework in Section 3. Clearly, the θ must also stay within the subset of general USTs that are relevant for the given

XSugar specification; otherwise, we will lose soundness of our static validation algorithm and the runtime behavior of unparsing will be undefined.

An example of a useful UST transformation is a function on the employee records from Section 5.1 that sorts the employees according to their SSNs. Also, rounding the salaries into the nearest whole dollar amount is an idempotent transformation.

The concrete design of a syntax for UST transformations is left as future work, but inspiration is available in [18] and [15].

7 Conclusion

We have presented the XSugar system, which allows specification of languages with dual syntax—one of which is XML-based—and provides translations in both directions. Moreover, we have presented techniques for statically checking reversibility of an XSugar specification and validity of the output in the direction that generates XML. Finally, we have conducted a number of experiments by applying the system to various existing languages with dual syntax. Of course, XSugar does not support all imaginable transformations; nonetheless, all dual syntaxes that we have encountered fit into our model. We conclude that XSugar provides sufficient expressiveness and useful static guarantees, and at the same time allows concise specifications making it a practically useful system.

References

[1] Sergei Abramov and Robert Glück. Principles of inverse computation and the universal resolving algorithm. In *The essence of computation: complexity, analysis, transformation*, pages 269–295. Springer-Verlag, 2002.

[2] Nitesh Ambastha and Tahir Hashmi. Xqueeze, 2005. `http://xqueeze.sourceforge.net/`.

[3] Vidur Apparao et al. Document Object Model (DOM) level 1 specification, October 1998. W3C Recommendation. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[4] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Datatypes second edition, October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

[5] John Boyer. Canonical XML Version 1.0, March 2001. W3C Recommendation. `http://www.w3.org/TR/xml-c14n`.

[6] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, volume 4783 of *LNCS*. Springer-Verlag, July 2007.

[7] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221–231, June 2001.

[8] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML, January 1999. W3C Recommendation. `http://www.w3.org/TR/REC-xml-names/`.

[9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (third edition), February 2004. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

[10] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

[11] James Clark. RELAX NG compact syntax, November 2002. OASIS. `http://relaxng.org/compact.html`.

[12] James Clark and Makoto Murata. RELAX NG specification, December 2001. OASIS. `http://www.oasis-open.org/committees/relax-ng/`.

[13] Clear Methods, Inc. ConciseXML, 2005. `http://www.concisexml.org/`.

[14] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[15] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05*, January 2005.

[16] Vidar Bronken Gundersen and Zeger W. Hendrikse. BibTeXML, 2005. `http://bibtexml.sourceforge.net/`.

[17] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[18] Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 06*, September 2006.

[19] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, 2005. Presented at Programming Language Technologies for XML, PLAN-X '06.

[20] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[21] Christian Kirkegaard and Anders Møller. dk.brics.schematools, 2007. `http://www.brics.dk/schematools/`.

[22] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[23] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley, 2001.

[24] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, and Mary Fernandez. PADS/ML: A functional data description language. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07*, January 2007. `http://www.padsproj.org/`.

[25] Sean McGrath. *XML processing with Python*. Prentice Hall, 2000.

[26] Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

[27] Anders Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

[28] Anders Møller and Michael I. Schwartzbach. XML graphs in program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '07*, January 2007.

[29] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectionalising HaXML, 2005.

[30] Anders Møller. dk.brics.grammar – context-free grammars for Java. `http://www.brics.dk/grammar/`.

[31] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2007. `http://www.brics.dk/automaton/`.

[32] Daniel Parker. Presenting XML, 2005. `http://presentingxml.sourceforge.net/`.

[33] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures second edition, October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

[34] Unidex Inc. XFlat, 2005. `http://www.unidex.com/xflat.htm`.