Dual Syntax for XML Languages

Claus Brabrand, Anders Møller*, and Michael I. Schwartzbach

BRICS**, 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.

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 [9]. 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 [8]. A main goal of providing the non-XML syntax is to maximize readability. As an example (taken from the RELAX NG documentation),

* Supported by the Carlsberg Foundation contract number 04-0080.
** Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

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 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 programming 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 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 checkable conditions for reversibility.
- We show that it is possible to statically guarantee validity of output for the translation to XML.
- Using a prototype implementation, we evaluate the approach on a number of real-world examples: RELAX NG, XFlat [18], BibTeXML [12], and XSugar itself.

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.

1.2 Related Work

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 [17] 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 [18] 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 [11] translates data into other representations, including XML. It is focused on streams of data items, which are described using a sophisticated calculus that include dependent types and computations—thus going beyond context-free parsing. PADS also differs from XSugar in that its translations are not automatically reversible. The paper [16] presents a framework for programming reversible translations between two XML languages, but does not consider the case of parsing or generating alternative syntax.

Several projects, such as [10, 2, 14], 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.

2.1 Example: Student Information

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. A valid document is the following:

```
<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
```

That is, each student corresponds to one line. The name is written first, then the email address in parentheses, 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/" ;

Alpha  = [a-zA-Z_] ;
Name   = <Alpha>+(" "<Alpha>+)* ;
Email  = <Alpha>+"@"<Alpha>+("."<Alpha>+)+ ;
Id     = [0-9]{8} ;
NL     = "\r"*"\n" ;

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

persons : [person p] [NL] [persons more] = _ [person p] _ [persons more] ;
        : = _ ;

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

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

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

(If the nonterminal is omitted, the one from the preceding production is assumed.) The α part is generally a sequence of *items* of the form `[`T *name*`]` or `[`T`]`, where T is either a nonterminal or a regular expression name, and of literals such as `(` and `)` above. Additionally, the special character `_` is used for describing whitespace, which we return to later. 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 (as `sid=[Id id]` in the example) and in element content (as `[Email email]`, for example). Also the nonterminal or regular expression name associated with a given item name must be the same in α and β. We use the convention that regular expression names start with a capital letter, and nonterminals start with a lower case letter. Special characters (such as `=` and `;`) can be escaped with a backslash notation or Unicode escapes as in Java. XML character references can also be used in XML templates.

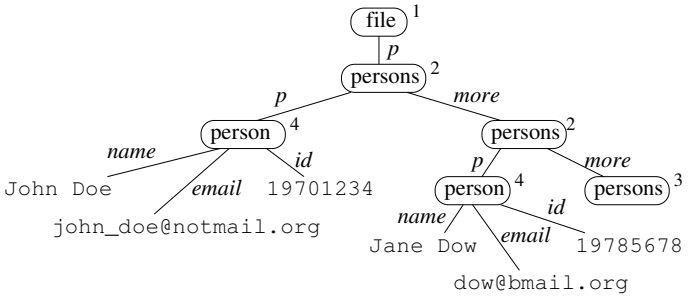

Fig. 1. UST for the student information document.

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. Notice that 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 *conjunctive* productions: In a production where the delimiter `:&` appears in place of `:`, the α part is conjunctive, meaning that it matches any permutation of the constituents. Similarly, using the delimiter `=&` in place of `=` makes the β part conjunctive. We show a use of conjunctive productions in Section 5.4.

2.2 Transforming via Unifying Syntax Trees

An XSugar specification $\mathcal{S}$ additionally 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 an index, which we will need later. As an example, the UST corresponding to the example student information document is shown in Figure 1.

Assume that we want to transform a text x from the non-XML syntax to the XML syntax. This is done in two steps: (1) we first *parse* the text x according to $\mathcal{S}_\alpha$, yielding a UST u; (2) we 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 are described in the following.

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 an ordinary context-free-grammar parser on x and $\mathcal{S}_i$, yielding an ordinary parse tree t. (If $\mathcal{S}_i$ is ambiguous, t is chosen arbitrarily among the possibilities; we discuss ambiguity further in Section 3.) From this 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.

Note that all parse tree nodes corresponding to literals or unnamed items are ignored in the construction.

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 characters). Similarly, `__` refers to `WHITESPACE`, which represents *nonempty* strings of whitespace.

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 [3]: (1) whitespace inside tags (but outside attribute values) is reduced to a minimum; (2) the attributes in each start tag are sorted lexicographically; (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 where possible; (6) XML comments, XML declarations, and DOCTYPEs are removed; and (7) in every start tag, all namespace declarations that are used in the tag are inserted explicitly, and prefixes are renamed to coincide with those chosen in $\mathcal{S}$.

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. 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;

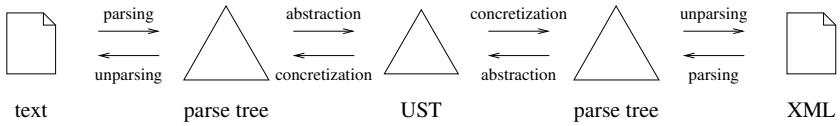

Fig. 2. The transformation process.

- 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 picking arbitrary representatives. This makes sense since such items describe information that only occurs in one of the two syntaxes.

Once we have the parse tree t, the resulting text x is simply the concatenation of the text in the leaves.

One technical issue remains: We escape and unescape special XML characters to ensure that, for example, the character `<` in non-XML corresponds to `<` in XML.

Figure 2 shows the complete transformation process with parsing, abstraction, concretization, and unparsing.

3 Reversibility

Having two syntaxes for a document poses a problem in that one would like to maintain a document in only one of the two syntaxes. However, since the two syntaxes are to represent the same logical information, the ideal solution would be to be able to move freely between them without loss of information. This imposes some static demands on a stylesheet.

In order to achieve this goal, a stylesheet needs to be *reversible* meaning that a roundtrip to the other syntactic alternative and back should yield the exact same document. In practice, however, this is too strong a property to work with due to ignorable information, such as whitespace and comments. For that reason, it is convenient to work with a weaker reversibility property that takes such things into account.

The notion of ignorable information is precisely what is captured by the unnamed items in an XSugar stylesheet, which in this way explicitly annotates certain information as ignorable. Such information is not to be recorded and injected into the other syntactic alternative; as explained above the parser discards such information and the unparser in turn invents representatives.

Since the transformations are conducted in the same way for both syntaxes, we only need to be able to check that (1) parsing/unparsing to and from ordinary parse trees is bijective modulo ignorable information and that (2) abstraction/concretization to and from USTs is bijective modulo ignorable information.

The parsing/unparsing check is equivalent to deciding whether a context-free grammar is ambiguous modulo ignorable constituents, which is of course undecidable. However, we deal with this issue by relying on a static analysis based on regular approximations of context-free grammars [4]. The analysis conversatively approximates the decision problem in that if it says that a grammar is unambiguous then this is indeed the case, but for certain grammars, the analysis will be unable to give a definitive answer. This is reminiscent of the LR(k) and LALR(k) ambiguity checks in Yacc/Bison, but with built-in support for ignorable constituents. Unambiguity is, aside from reversibility issues, a desirable property for a grammar, so that there can be no misunderstandings as to how a string is interpreted by a parser.

As for the second check, recall that all UST tree nodes are annotated with their production indices and all edges to subtrees are labeled with item names. This means that we simply have to check that all named items are used exactly once on the other side, so that no non-ignorable information is ever thrown away by the abstraction.

4 Static Validation

Consider the typical situation where an XML language, described by some schema formalism, has been given an alternative syntax. An obvious *validation* check is that the translations of alternative documents will always result in valid XML documents.

XSugar performs a *static analysis* that conservatively approximates this check. When the analysis reports success, it is guaranteed that syntactically correct input always results in valid output.

The dual validation check 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 to allow the XSugar translation. However, the inclusion test between context-free grammars is of course undecidable, 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. This is an interesting problem that at present is left for future work. The dual coverage check is just the opposite inclusion check between the two context-free grammars. Note that it will often be the case that the alternative syntax is simply defined by the XSugar specification. In that situation, both the non-XML to XML coverage checks and the XML to non-XML validation checks become trivial.

Our static analysis is based on previous results [5, 7, 13], where the concept of *summary graph* is used to model sets of XML documents. We have an algorithm that is able statically to check that every document described by a summary graph is valid according to a DSD2 schema [15]. Through embeddings, this tech-

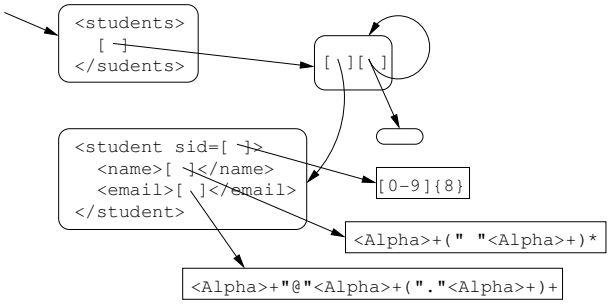

Fig. 3. Summary graph for the student information example.

nique also works for DTD and XML Schema. Note that a summary graph is simply a special representation of a regular tree language [6].

From an XSugar specification, it is simple to extract a summary graph that describes all XML documents that can be generated by the β productions: Each right-hand side becomes a summary graph node and the edges reflect the possible derivations of nonterminal and terminal items. For the student information example, this looks as shown in Figure 3. The static validation is then performed by checking this summary graph against the given XML schema [7]. This is further exemplified in Section 5.3.

5 Evaluation

We have implemented a fully functional prototype of the XSugar tool, which is available for download from `http://www.brics.dk/xsugar/`. The underlying parser is a variation of Earley's algorithm that builds a UST directly without the intermediate ordinary parse tree, has explicit support for regular expression items, and allows the conjunctive productions explained in Section 2.1. The tool also performs the static validation described in Section 4, by means of the summary graph validator component from the JWIG project [7].

In the following, 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 source of the these XSugar specifications are available at the URL mentioned above, along with examples of input and output documents.

5.1 XFlat

The XFlat system [18] 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> ;

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

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

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.

In other respects, the XSugar tool is more general. 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 allows an alternative syntax, which may be captured 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 non-empty 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 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 sequence. These techniques may in general be necessary, but this particular example requires by far the most complex unfoldings that we have yet encountered.

On the RELAX NG site, the 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 [12] 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 detailed 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. 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:

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

authors : [name n] = <bibxml:author> _ [name n] _ </bibxml:author> ;
        : [name n] [AND] [authors as] =
          <bibxml:author> _ [name n] _ </bibxml:author> _ [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 conjunctive 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] _
            </bibxml:article> _
          </bibxml:entry> ;

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 conjunctive 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 normalized representation.

Static validation of the generated XML documents is for this substantial example performed in 6 seconds (on a standard PC). The analysis discovered 4 true errors in the definition of the BiBTeX translation (despite our best efforts at defining it correctly), which were subsequently corrected. No false errors were reported.

5.4 XSugar

The final example applies XSugar to itself, by providing an XML syntax inspired by XSLT. Apart from the amusement of self-application, this example demonstrates the use of another feature. The production for the dual syntax for literal XML elements looks as follows:

```
element : "<" _ [qname q] _ [attributes as] _ ">"
            _ [xml x] _
          "</" _ [qname q] _ ">" =
          <xsg:element name=[qname q]> _
            [attributes as] _ [xml x] _
          </xsg:element> ;
```

We extend the XSugar language by allowing the identifier `q` to appear twice in the rule for the non-XML syntax. When translating from XML to non-XML syntax, the corresponding string is copied to the two locations, and in the other direction the parser checks that the two USTs generate the same output (and picks either one of them). This ability to match subterms for equality during parsing of course means that we go beyond context-free languages, while maintaining the functionality and guarantees of the XSugar tool.

6 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; however, 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. John Boyer. Canonical XML Version 1.0, March 2001. W3C Recommendation. `http://www.w3.org/TR/xml-c14n`.
4. Claus Brabrand and Anders Møller. Analyzing ambiguity of context-free grammars, 2005. In preparation.
5. 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.
6. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.
7. 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, November 2003.
8. James Clark. RELAX NG compact syntax, November 2002. OASIS. `http://relaxng.org/compact.html`.
9. James Clark and Makoto Murata. RELAX NG specification, December 2001. OASIS. `http://www.oasis-open.org/committees/relax-ng/`.
10. Clear Methods, Inc. ConciseXML, 2005. `http://www.concisexml.org/`.
11. Kathleen Fisher et al. PADS: Processing Arbitrary Data Streams, 2005. `http://www.padsproj.org/`.
12. Vidar Bronken Gundersen and Zeger W. Hendrikse. BibTeXML, 2005. `http://bibtexml.sourceforge.net/`.
13. 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.
14. Sean McGrath. *XML processing with Python*. Prentice Hall, 2000.
15. 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/`.
16. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectionalising HaXML, 2005.
17. Daniel Parker. Presenting XML, 2005. `http://presentingxml.sourceforge.net/`.
18. Unidex Inc. XFlat, 2005. `http://www.unidex.com/xflat.htm`.