XML and Multilingual Document Authoring: Convergent Trends

Marc Dymetman **Veronika Lux**
Xerox Research Centre Europe
6, chemin de Maupertuis
38240 Meylan, France
{dymetman,lux}@xrce.xerox.com

Aarne Ranta
Department of Computing Science
Chalmers University of Technology
and Göteborg University
S-412 96 Göteborg, Sweden
aarne@cs.chalmers.se

Abstract

Typical approaches to XML authoring view a XML document as a mixture of structure (the tags) and surface (text between the tags). We advocate a radical approach where the surface disappears from the XML document altogether to be handled exclusively by rendering mechanisms. This move is based on the view that the author's choices when authoring XML documents are best seen as language-neutral semantic decisions, that the structure can then be viewed as interlingual content, and that the textual output should be derived from this content by language-specific realization mechanisms, thus assimilating XML authoring to Multilingual Document Authoring. However, standard XML tools have important limitations when used for such a purpose: (1) they are weak at propagating semantic dependencies between different parts of the structure, and, (2) current XML rendering tools are ill-suited for handling the grammatical combination of textual units. We present two related proposals for overcoming these limitations: one (GF) originating in the tradition of mathematical proof editors and constructive type theory, the other (IG), a specialization of Definite Clause Grammars strongly inspired by GF.

1 Introduction

The typical approach to XML authoring views an XML document as a mixture of tree-like *structure*, expressed through balanced labelled parentheses (the tags), and of *surface*, expressed through free text interspersed between the tags (PCDATA). A Document Type Definition (DTD) is roughly similar to a context-free grammar[1] with exactly one predefined terminal. It defines a set of well-formed structures, that is, a *language over trees*, where each nonterminal node can dominate either the empty string, or a sequence of occurrences of nonterminal nodes and of the terminal node `pcdata`. The terminal `pcdata` has a special status: it can in turn dominate any character string (subject to certain restrictions on the characters allowed). Authoring is typically seen as a top-down interactive process of step-wise refinement of the root nonterminal (corresponding to the whole document) where the author iteratively chooses a rule for expanding a nonterminal already present in the tree,[2] and where in addition the author can choose an arbitrary sequence of characters (roughly) for expanding the `pcdata` node.

One can observe the following trends in the XML world:

- A move towards more typing of the surface: *Schemas* (W3C, 1999a), which are an influential proposal for the replacement of DTD's, provide for types such as `float`, `boolean`, `uri`, etc., instead of the single type `pcdata`;
- A move, already constitutive of the main purpose of XML as opposed to HTML for instance, towards clearer separation between content and form, where the original XML document is responsible for content, and powerful styling mechanisms (e.g. XSLT (W3C, 1999b)) are available for rendering the document to the end-user.

We advocate an approach in which these two moves are radicalized in the following ways:

Strongly typed, surface-free XML documents. The whole content of the document is a tree where each node is labelled and typed. For internal nodes, the type is just the usual nonterminal name (or category), and the label is a name for the expansion chosen for this nonterminal, that is, an identifier of which rule was chosen to expand this nonterminal. For leaves, the type is a semantically specific category such as `Integer`, `Animal`, etc., and the label is a specific concept of this type, such as `three` or `dog`.[3]

Styling responsible for producing the text itself. The styling mechanism is not only responsible for rendering the layout of the text (typography, order and presentation of the elements), but also for producing *the text itself* from the document content.

What are the motivations behind this proposal?

Authoring choices carry language-independent meaning. First, let us note that the expansion choices

[1] But see (Prescod, 1998) for an interesting discussion of the differences.

[2] We are ignoring here the aspects of this process relating to the regular nature of the right-hand sides of rules, but these particulars are unessential to the main argument.

[3] Note that Integer is of "logical type" e, whereas Animal is of logical type $\langle e, t\rangle$: there is no restriction on the denotational status of leaves.

```
<!ELEMENT Risk (Caution | Warning) >        risk-rule1:   Risk --> Caution
                                            risk-rule2:   Risk --> Warning

<!ELEMENT Caution ( ... | ... | ... ) >     caution-rule1:   Caution --> ...
                                            caution-rule2:   Caution --> ...
                                            caution-rule3:   Caution --> ...
...
```

Figure 1: Context-free rules (shown on the right) corresponding to the aircraft DTD (shown on the left); for illustration purposes, we have assumed that there are in turn three semantic varieties of cautions. The rule identifier on the left can be seen as a semantic label for each expansion choice (in practice, the rule identifiers are given mnemonic names directly related to their meaning).

made during the authoring of an XML document generally carry *language-independent meaning*. For instance, the DTD for an aircraft maintenance manual might be legally required to distinguish between risk instructions of two kinds: `caution` (risk related to material damages) and `warning` (risk to the operator). Or a DTD describing a personal list of contacts might provide a choice of gender (`male`, `female`), title (`dr`, `prof`, `default`), country (`ger`, `fra`,...), etc. Each such authoring choice, which formally consists in selecting among different rules for expanding the same nonterminal (see Figure 1), corresponds to a *semantic decision* which is independent of the language chosen for expressing the document. A given DTD has an associated *expressive space* of tree structures which fall under its *explicit* control, and the author is situating herself in this space through top-down expansion choices. There is then a tension between on the one hand these explicitely controlled choices, which should be rendered differently in different languages (thus `ger` as *Germany*, *Allemagne*, *Deutschland* ..., and `Warning` by a paragraph starting with *Warning!* ...; *Attention, Danger!* ...; *Achtung, Lebensgefahr!* ...), and on the other hand the uncontrolled inclusion in the XML document of free PCDATA strings, which are written in a specific language.

Surface-free XML documents. We propose to completely remove these surface strings from the XML document, and replace them with explicit meaning labels.[4] The tree structure of the document then becomes the sole repository of content, and can be viewed as a kind of *interlingua* for describing a point in the expressive space of the DTD (a strongly domain-dependent space); it is then the responsability of the language-specific rendering mechanisms to "display" such content in each individual language where the document is needed.

XML and Multilingual Document Authoring. In this conception, XML authoring has a strong connection to the enterprise of *Multilingual Document Authoring* in which the author is guided in the specification of the document content, and where the system is responsible for generating from this content textual output in several languages simultaneously (see (Power and Scott, 1998; Hartley and Paris, 1997; Coch, 1996)).

Now there are some obvious problems with this view, due to the current limitations of XML tools.

Limitations of XML for multilingual document authoring. The first, possibly most serious, limitation originates in the fact that a standard DTD is severely restricted in the semantic dependencies it can express between two subtrees in the document structure. Thus, if in the description of a contact, a city of residence is included, one may want to constrain such an information depending on the country of residence; or, in the aircraft maintenance manual example, one might want to automatically include some warning in case a dangerous chemical is mentioned somewhere else in the document. Because DTD's are essentially of context-free expressive power, the only communication between a subtree and its environment has to be mediated through the name of the nonterminal rooting this subtree (for instance the nonterminal `Country`), which presents a bottleneck to information flow.

The second limitation comes from the fact that the current styling tools for rendering an XML document, such as CSS (Cascading Style Sheets), which are a strictly layout-oriented language, or XSLT (XSL transformation language), which is a more generic tool for transforming an XML document into another one (such as a display-oriented HTML file) are poorly adapted to linguistic processing. In particular, it seems difficult in such formalisms to express such basic grammatical facts as number or gender agreement. But such problems become central as soon as semantic elements corresponding to textual units below the sentence level have to be combined and rendered linguistically.

We will present two related proposals for overcoming these limitations. The first, the *Grammatical Framework (GF)*(Ranta, 2000), originates in constructive type-theory (Martin-Löf, 1984; Ranta, 1994) and in mathematical proof editors (Magnusson and Nordström, 1994). The second, *Interaction Grammars (IG)*, is a specialization of Definite Clause Grammars strongly inspired by GF. The two approaches present certain formal differences that will not be examined in detail in this paper,

[4]There are authoring situations in which it may be necessary for the user to introduce new semantic labels corresponding to expressive needs not foreseen by the creator of the original DTD. To handle such situations, it is useful to view the DTD's as open-ended objects to which new semantic labels and types can be added at authoring time.

but they share a number of important assumptions:

- The semantic representations are *strongly typed trees*, and rich dependencies between subtrees can be specified;
- The abstract tree is *independent* of the different textual realization languages;
- The surface realization in each language is obtained by a *semantics-driven compositional process*; that is, the surface realizations are constructed by a bottom-up recursive process which associates surface realizations to abstract tree nodes by recursively combining the realizations of daugther nodes to obtain the realization of the mother node.
- The grammars are *reversible*, that is, can be used both for generation and for parsing;
- The authoring process is an interactive process of repeatedly asking the author to further specify nodes in the abstract tree of which only the type is known at the point of interaction (*type refinement*). This process is mediated through text in the language of the author, showing the types to be refined as specially highlighted textual units.

2 GF — the Grammatical Framework

The Grammatical Framework (GF; (Ranta, 2000)) is a special-purpose programming language combining *constructive type theory* with an annotation language for concrete syntax. A *grammar*, in the sense of GF, defines, on one hand, an *abstract syntax* (a system of types and typed syntax trees), and on the other hand, a mapping of the abstract syntax into a *concrete syntax*. The abstract syntax has *category* declarations, such as

```
cat Country ; cat City ;
```

and *combinator* (or *function*) declarations, such as

```
fun Ger : Country ; fun Fra : Country ;
fun Ham : City ; fun Par : City ;
fun cap : Country -> City ;
```

The type of a combinator can be either a basic type, such as the type `City` of the combinator `Ham`, or a function type, such as the type of the combinator `cap`. Syntax trees formed by combinators of function types are complex functional terms, such as

```
cap Fra
```

of type `City`.

The concrete syntax part of a GF grammar gives *linearization* rules, which assign strings (or, in general, more complex linguistic objects) to syntax trees. For the abstract syntax above, we may have

```
lin Ger =3D "Germany" ; lin Fra =3D "France" ;
lin Ham =3D "Hamburg" ; lin Par =3D "Paris" ;
lin cap Co =3D "the capital of" ++ Co ;
```

Thus the linearization of `cap Fra` is

```
the capital of France
```

2.1 GF in XML

Functional terms have a straightforward encoding in XML, representing a term of the form

$$f\ a_1\ \ldots\ a_n$$

by the XML object

$$\texttt{<}f\texttt{>}\ a'_1\ \ldots\ a'_n\ \texttt{</}f\texttt{>}$$

where each a'_i is the encoding of a_i. In this encoding, `cap Fra` is

```
<cap>
 <Fra>
 </Fra>
</cap>
```

The simple encoding does not pay attention to the types of the objects, and has no interesting DTD. To express type distinctions, we will hence use a slightly more complicated representation, in which the category and combinator declarations of GF are represented as DTDs in XML, so that GF *type checking* becomes equivalent with XML *validation*. The representation of the GF grammar of the previous section is the DTD

```
<!ELEMENT Country (Ger | Fra) >
<!ELEMENT Ger EMPTY >
<!ELEMENT Fra EMPTY >
<!ELEMENT City (Ham | Par | (cap,Country))>
<!ELEMENT Ham EMPTY >
<!ELEMENT Par EMPTY >
<!ELEMENT cap EMPTY >
```

In this DTD, each category is represented as an ELEMENT definition, listing all combinators producing trees of that category. The combinators themselves are represented as EMPTY elements. The XML representation of the capital of France is

```
<City>
 <cap />
 <Country>
  <Fra />
 </Country>
</City>
```

which is a valid XML object w.r.t. the given DTD.

The latter encoding of GF in XML enjoys two important properties:

- All well-typed GF trees are represented by valid XML objects.
- An XML represents a unique GF tree.

The first property guarantees that *type checking* in the sense of GF (and type theory) can be used for *validation* of XML objects. The second property guarantees that GF objects can be stored in the XML format. (The second property is already guaranteed by the simpler encoding, which ignores types.)

Other properties one would desire are the following:

- All valid XML objects represent well-typed GF trees.
- A DTD represents a unique GF abstract grammar.

These properties cannot be satisfied, in general. The reason is that GF grammars may contain *dependent types*, i.e. types depending on objects. We will return to this notion shortly. But let us first consider the use of GF for multilingual generation.

2.2 Multilingual generation in GF

Multilingual generation in GF is based on *parallel grammars*: two (or more) GF grammars are parallel, if they have the same abstract syntax. They may differ in concrete syntax. A grammar parallel to the one above is defined by the concrete syntax

```
param Case =3D nom | gen ;

oper nom1 : Str -> Case =3D> Str =3D
  \s -> tbl {{nom} =3D> s, {gen} -> s+"n"} ;
oper nom2 : Str -> Case =3D> Str =3D
  \s -> tbl
    {{nom} =3D> s+"ki", {gen} -> s+"gin"} ;

lincat Country =3D Case =3D> Str ;
lincat City    =3D Case =3D> Str ;

lin Ger =3D nom1 "Saksa" ;
lin Fra =3D nom1 "Ranska" ;
lin Ham =3D nom1 "Hampuri" ;
lin Par =3D nom1 "Pariisi" ;
lin cap Co =3D
  tbl {c =3D> Co!gen ++
    nom2 "pääkaupun"!c} ;
```

This grammar renders GF objects in Finnish. In addition to linearization rules, it has rules introducing parameters and operations, and rules defining the *linearization types* corresponding to basic types: the linearization type of `Country`, for instance is not just string (`Str`), but a function from cases to strings.

Not only the linearization rules proper, but also parameters and linearization types vary a lot from one language to another. In our example, we have the parametre of case with two values (in larger grammars for Finnish, as many as 16 may be required!), and two patterns for inflecting Finnish nouns. The syntax tree `cap Fra` produces the strings

```
Ranskan pääkaupunki
Ranskan pääkaupungin
```

which are the nominative and the genitive form, respectively.

2.3 Dependent types

DTDs in XML are capable of representing *simple types*, i.e. types without dependencies. Even a simple type system can contribute a lot to the *semantic control* of documents. For instance, the above grammar permits the formation of the English noun phrase

```
the capital of France
```

but not of

```
the capital of Paris
```

Both of these expressions would be well-formed w.r.t. an "ordinary" grammar, in which both France and Paris would be classified simply as noun phrases.

Dependent types are types depending on objects of other types. An example is the following alternative declaration of `Country` and `City`:

```
cat Country ; cat City (Co:Country) ;
```

Under this definition, there are no objects of type `City` (which is no longer a well-formed type), but of types `City Ger` and `City Fra`. Thus we define e.g.

```
fun Ham : City Ger ; fun Par : City Fra ;
fun cap : (Co:Country) -> City Co ;
```

Observe the use of the variable `Co` in the type of the combinator capital: the variable is bound to the argument type and then used in the value type. The capital of a country is by definition a city of the same country. This involves a generalization of function types with dependent types.

Now consider a simplified format of postal addresses: an address is a pair of a country and a city. The GF rule is either

```
fun addr : Country -> City -> Address ;
lin addr Co C =3D C ++ "," ++ Co ;
```

using simple types or

```
fun addr :
     (Co:Country) -> City Co -> Address ;
lin addr Co C =3D C ++ "," ++ Co ;
```

using dependent types. The invalid address

```
Hamburg, France
```

is well-typed by the former definition but not by the latter. Using the latter definition gives a simple mechanism of *semantic control* of addresses. The same idea can obviously be extended to full addresses with street names and numbers. Such dependencies cannot, however, be expressed in DTDs: both of the address rules above correspond to one and the same ELEMENT definition,

```
<!ELEMENT Address (addr, Country, City) >
```

This example also shows that XML validity is not enough for GF well-formedness: the object

```
<Address>
 <addr />
 <Country>
  <Fra />
 </Country>
 <City>
  <Ham />
 </City>
</Address>
```

is valid w.r.t. the DTD, but the corresponding GF object

```
addr Fra Ham
```

is not well-typed.

2.4 Computation rules

In addition to categories and combinators, GF grammars may contain *definitions*, such as

```
def cap Fra =3D Par ;
```

Definitions belong to the abstract syntax. They define a *normal form* for syntax trees (recursively replace defienda by definientes), as well as a *paraphrase* relation (sameness of normal form). These notions are, of course reflected in the concrete syntax: the addresses

```
the capital of France, France
Paris, France
```

are paraphrases, and the latter is the normal form of the former.

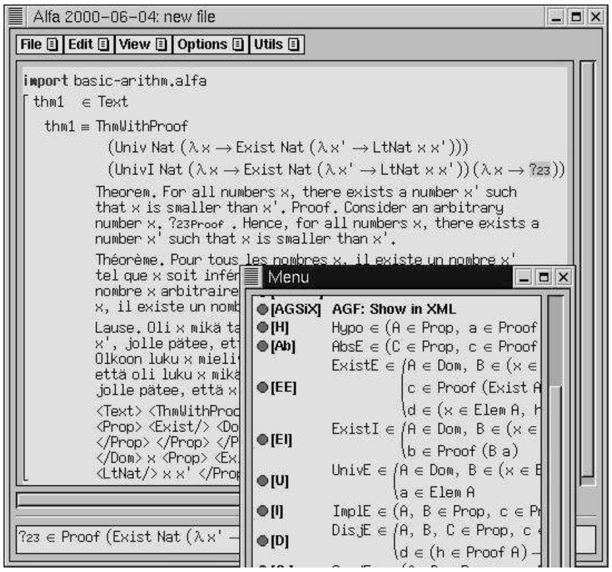

Figure 2: GF session for editing a mathematical proof text.

2.5 GF editing tools

An editing tool has been implemented for GF, using *metavariables* to represent yet undefined parts of expressions. The user can work on any metavariable, in various different ways, e.g.

- by choosing a combinator from a menu,
- by entering a string that is parsed,
- by reading a previously defined object from a file,
- by using an automatic search of suitable instantiations.

These functionalities and their metatheory have been used for about a decade in a number of syntax editors for constructive type theory, usually known as *proof editors* (Magnusson and Nordström, 1994). From this point of view, the GF editor is essentially a proof editor together with supplementary views, provided by the concrete syntax. The current implementation of GF is a plugin module of the proof editor Alfa (Hallgren, 2000). The window dump in Figure 2 shows a GF session editing a mathematical proof. Five views are provided: abstract syntax in type-theoretical notation, English, French, Finnish, and XML. One metavariable is seen, expecting the user to find a `Proof` of the proposition that there exists a number x' such that x is smaller than x', where x is an arbitrary number given in the context (for the sake of Universal Introduction).

3 IG : Interaction Grammars

We have just described an approach to solving the limitations of usual XML tools for multilingual document authoring which originates in the tradition of constructive type-theory and mathematical proof editors. We will now sketch an approach strongly inspired by GF but which formally is more in the tradition of logic-programming based unification grammars, and which is currently under development at Xerox Research Centre Europe (see (Brun et al., 2000) for a more extended description of this project).

Definite Clause Grammars, or DCG's, (Pereira and Warren, 1980), are possibly the simplest unification-based extension of context-free grammars, and have good reversibility properties which make them adapted both to parsing and to generation. A typical view of what a DCG rule looks like is the following:[5]

```
a(a1(B,C,...)) -->
       <text1>,
       b(B),
       <text2>,
       c(C),
       <text3>,
       ...
       {constraints(B,C,...)}.
```

This rule expresses the fact that (1) some abstract structure `a1(B,C,...)` is in category `a` if the structure `B` is in category `b`, the structure `C` in category `c`, ..., and furthermore a certain number of constraints are satisfied by the structures `B`, `C`, ...; (2) if the structures `B`, `C`, ... can be "rendered" by character strings `StringB`, `StringC`, ..., then the structure `a1(B,C,...)` can be rendered by the string obtained by concatenating the text `<text1>` (that is, a certain constant sequence of terminals), then `StringB`, then `<text2>`, then `StringC`, etc.

In this formalism, a grammar for generating English addresses (see preceding section) might look like:

[5] Reminder: according to the usual logic programming conventions, lowercase letters denote predicates and functors, whereas uppercase letters denote metavariables that will be instantiated with terms.

```
address(addr(Co,C)) --> city(C), ",",
                        country(Co).
country(fra) --> "France".
country(ger) --> "Germany".
city(par) --> "Paris".
city(cap(Co)) --> "the capital of",
                  country(Co).
```

The analogies with the GF grammars of the previous section are clear. What is traditionally called a category (or nonterminal, or predicate) in the logic programming terminology, can also be seen as a type (`address`, `country`, `city`) and functors such as `ger`, `par`, `addr`, `cap` can be seen as combinators.

If, in this DCG, we "forget" all the constant strings by replacing them with the empty string, we obtain the following "abstract grammar":

```
address(addr(Co,C)) --> city(C), country(Co).
country(fra) --> [].
country(ger) --> [].
city(par) --> [].
city(cap(Co)) --> country(Co).
```

which is in fact equivalent to the definite clause *program*:[6]

```
address(addr(Co,C)) :- city(C), country(Co).
country(fra).
country(ger).
city(par).
city(cap(Co)) :- country(Co).
```

This program is language-independent and recursively defines a set of well-formed trees to which it assigns types (thus `cap(fra)` is a well-formed tree of type `city`).

As they stand, such definite clause grammars and programs, although suitable for simple generation tasks, are not directly adapted for the process of interactive multilingual document authoring. In order to make them more appropriate for that task, we need to specialize and adapt DCGs in the way that we now describe.

Parallel grammars. The first move is to allow for parallel English, French, ..., grammars, which all have the same underlying abstract grammar (program). So in addition to the English grammar given above, we have the French grammar:

```
address(addr(Co,C)) --> city(C), ",",
                        country(Co).
country(fra) --> "la France".
country(ger) --> "l'Allemagne".
city(par) --> "Paris".
city(cap(Co)) --> "la capitale de",
                  country(Co).
```

[6]In the sense that rewriting the nonterminal goal `address(addr(Co,C))` to the empty string in the DCG is equivalent to proving the goal `address(addr(Co,C))` in the program (Deransart and Maluszynski, 1993).

Dependent Categories. The grammars we have given are deficient in one important respect: there is no dependency between the city and the country in the same address. In order to remedy this problem, a standard logic programming move would be to reformulate the abstract grammar (and similarly for the language-dependent ones) as:

```
address(addr(Co,C)) --> city(C,Co),
                        country(Co).
country(fra) --> [].
country(ger) --> [].
city(par,fra) --> [].
city(cap(Co),Co) --> country(Co).
```

The expression `city(C,Co)` is usually read as the relation "C is a city of Co", which is fine for computational purposes, but this reading obscures the notion that the object `C` is being *typed* as a `city`; more precisely, it is being typed as a `city` of `Co`. In order to make this reading more apparent, we will write the grammar as:

```
address(addr(Co,C)) --> city_Co(C),
                        country(Co).
country(fra) --> [].
country(ger) --> [].
city_fra(par) --> [].
city_Co(cap(Co)) --> country(Co).
```

That is, we allow the categories to be indexed by terms (a move which is a kind of "currying" of a relation into a type for its first argument). Dependent categories are similar to the dependent types of constructive type theory.

Heterogeneous trees. Natural language authoring is different from natural language generation in one crucial respect. Whenever the abstract tree to be generated is incomplete (for instance the tree `cap(Co)`), that is, has some leaves which are yet uninstantiated variables, the generation process should not proceed with nondeterministically enumerating texts for all the possible instantiations of the initial incomplete structure. Instead it should display to the author as much of the text as it can in its present "knowledge state", and enter into an interaction with the author to allow her to further refine the incomplete structure, that is, to further instantiate some of the uninstantiated leaves. To this purpose, it is useful to introduce along with the usual combinators (`addr`, `fra`, `cap`, etc.) new combinators of arity 0 called *typenames*, which are notated **type**, and are of type `type`. These combinators are allowed to stand as leaves (e.g. in the tree `cap(`**country**`)`) and the trees thus obtained are said to be *heterogeneous*. The typenames are treated by the text generation process as if they were standard semantic units, that is, they are associated with text units which are generated "at their proper place" in the generated output. These text units are specially phrased and highlighted to indicate to the author that some choice has to be made to refine the underlying type (e.g. obtaining the text “la capitale de PAYS"). This choice has the effect of further instantiating the incomplete tree with “true" combinators, and the generation process is iterated.

Extended semantics-driven compositionality. The simple DCG view presented at the beginning of this section sees the process of generating text from an abstract structure as basically a compositional process on strings, that is, a process where strings are recursively associated with subtrees and concatenated to produce strings at the next subtree level. But such a direct process of constructing strings has well-known limitations when the semantic and syntactic levels do not have such a direct correspondence (simple example: ordering a list of modifiers around a noun). We are currently experimenting with a powerful extension of string compositionality where the objects compositionally associated with abstract subtrees are not strings, but syntactic representations with rich internal structure. The text itself is obtained from the syntactic representation associated with the total tree by simply enumerating its leaves.

The picture we get of an IG grammar is finally the following:

```
a_{D,...}(a1(B,C,...))-Syn -->
    b_{E,...}(B)-SynB,
    c_{F,...}(C)-SynC,
    ...
    {constraints(B,C,...,D,E,F,...)},
    {compose_english(SynB, SynC, Syn)}.
```

The rule shown is a rule for English: the syntactic representations are language dependent; Parallel rules for the other languages are obtained by replacing the `compose_english` constraint (which is unique to this rule) by constraints appropriate to the other languages under consideration.

4 Conclusion

XML-based authoring tools are more and more widely used in the business community for supporting the production of technical documentation, controlling their quality and improving their reusability. In this paper, we have stressed the connections between these practices and current research in natural language generation and authoring. We have described two related formalisms which are proposals for removing some of the limitations of XML DTD's when used for the production of multilingual texts.

From a computational linguist's point of view, there might be little which seems novel or exciting in XML representations. Still XML has a great potential as a *lingua franca* and in driving a large community of users towards authoring practices where content is becoming more and more explicit. There may be a great opportunity here for researchers in natural language generation to connect to a growing source of applications.

Acknowledgements

Thanks for contributions, discussions and comments to Ken Beesley, Caroline Brun, Jean-Pierre Chanod, Marie-Hélène Corréard, Pierre Isabelle, Bengt Nordström, Sylvain Pogodalla and Annie Zaenen.

References

C. Brun, M. Dymetman, and V. Lux. 2000. Document structure and multilingual authoring. In *Proceedings of First International Natural Language Generation Conference (INLG '2000)*, Mitzpe Ramon, Israel, June.

J. Coch. 1996. Evaluating and comparing three text production techniques. In *Proceedings of the 16th International Conference on Computational Linguistics*.

P. Deransart and J. Maluszynski. 1993. *A Grammatical View of Logic Programming*. MIT Press.

Thomas Hallgren. 2000. Alfa Home Page. Available from `http://www.cs.chalmers.se/~hallgren/Alfa/`

A. Hartley and C. Paris. 1997. Multilingual document production: from support for translating to support for authoring. In *Machine Translation, Special Issue on New Tools for Human Translators*, pages 109–128.

L. Magnusson and B. Nordström. 1994. The ALF proof editor and its proof engine. In *Lecture Notes in Computer Science 806*. Springer.

P. Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Naples.

W. Pardi. 1999. *XML in Action*. Microsoft Press.

Fernando C. N. Pereira and David H. D. Warren. 1980. Definite clause grammars for language analysis. *Artificial Intelligence*, 13:231–278.

R. Power and D. Scott. 1998. Multilingual authoring using feedback texts. In *Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics*, pages 1053–1059.

P. Prescod. 1998. Formalizing SGML and XML Instances and Schemata with Forest Automata Theory. `http://www.prescod.net/forest/shorttut/`.

A. Ranta. 1994. *Type-Theoretical Grammar*. Oxford University Press.

Aarne Ranta. 2000. GF Work Page. Available from `http://www.cs.chalmers.se/~aarne/GF/pub/work-index/`

W3C, 1998. *Extensible Markup Language (XML) 1.0*, February. W3C recommendation.

W3C, 1999a. *XML Schema - Part 1: Structures, Part 2 : Datatypes* -, December. W3C Working draft.

W3C, 1999b. *XSL Transformations (XSLT)*, November. W3C recommendation.