Abstract

In this paper, we present a metagrammar system for tree-based grammars which differs from comparable existing approaches both linguistically and computationally.

Linguistically, the formalism we introduce is both expressive and extensible. In particular, we show that it supports the description and factorization both of trees and of tree descriptions; that it allows the synchronized description of several linguistic dimensions (e.g., syntax and semantics) and that it includes a sophisticated treatment of the interaction between inheritance and identifier naming.

Computationally, the production of a grammar from a metagrammar is handled using powerful and well-understood logic programming techniques. A metagrammar is viewed as an extended definite clause grammar and compiled using a virtual machine closely resembling the Warren’s Abstract Machine. The generation of the trees satisfying a given tree description is furthermore handled using a tree description solver.

The paper is structured as follows. We begin (section 2) by introducing the linguistic formalism used for describing and factorizing tree based grammars. We then sketch the logic programming techniques used by the metagrammar compiler (section 3). Section 4 presents some evaluation results concerning the use of the system for implementing different types of tree based grammars. Section 5 concludes with pointers for further research and improvements.

1 Linguistic formalism

As mentioned above, the XMG system produces a grammar from a linguistic meta-description called a metagrammar. This description is specified using the XMG metagrammar formalism which sup-
ports three main features:

1. the **reuse** of tree fragments
2. the **specialization** of fragments via inheritance
3. the **combination** of fragments by means of conjunctions and disjunctions

These features reflect the idea that a metagrammar should allow the description of two main axes: (i) the specification of elementary pieces of information (fragments), and (ii) the combination of these to represent alternative syntactic structures.

**Describing syntax**  In a tree-based metagrammar, the basic informational units to be handled are tree fragments. In the XMG formalism, these units are put into **classes**. A class associates a **name** with a **content**. At the syntactic level, a content is a tree description\(^2\). The tree descriptions supported by the XMG formalism are defined by the following tree description language:

\[
\text{Description} ::= x \rightarrow y | x \rightarrow^+ y | x \rightarrow^* y | x \prec y | x \prec^+ y | x \prec^* y | x[f:E] \quad (1)
\]

where \(x, y\) represent node variables, \(\rightarrow\) immediate dominance (\(x\) is directly above \(y\)), \(\rightarrow^+\) strict dominance (\(x\) is above \(y\)), \(\rightarrow^*\) large dominance (\(x\) is above or equal to \(y\)), \(\prec\) is immediate precedence, \(\prec^+\) strict precedence, and \(\prec^*\) large precedence\(^3\). \(x[f:E]\) constrains feature \(f\) with associated expression \(E\) on node \(x\) (a feature can for instance refer to the **syntactic category** of the node)\(^4\).

Tree fragments can furthermore be combined using **conjunction** and/or **disjunction**. These two operators allow the metagrammar designer to achieve a high degree of factorization. Moreover the XMG system also supports **inheritance** between classes, thus offering more **flexibility** and **structure sharing** by allowing one to reuse and specialize classes.

**Identifiers’ scope**  When describing a broad-coverage grammar, dealing with identifiers scope is a non-trivial issue.

In previous approaches to metagrammar compilation ((Candido, 1999), (Gaiffe et al., 2002)), node identifiers had global scope. When designing broad-coverage metagrammars however, such a strategy quickly reduces modularity and complexities of grammar maintenance. To start with, the grammar writer must remember each node name and its interpretation and in a large coverage grammar the number of these node names amounts to several hundreds. Further it is easy to use twice the same name erroneously or on the contrary, to mistype a name identifier, in both cases introducing errors in the metagrammar.

In XMG, identifiers are local to a class and can thus be reused freely. Global and semi-global (i.e., global to a subbranch in the inheritance hierarchy) naming is also supported however through a system of **import / export** inspired from **Object Oriented Programming**. When defining a class as being a sub-class of another one, the XMG user can specify which are the viewable identifiers (i.e. which identifiers have been exported in the super-class).

**Extension to semantics**  The XMG formalism further supports the integration in the grammar of semantic information. More generally, the language manages **dimensions** of descriptions so that the content of a class can consists of several elements belonging to different dimensions. Each dimension is then processed differently according to the output that is expected (trees, set of predicates, etc).

Currently, XMG includes a semantic representation language based on **Flat Semantics** (see (Gardent and Kallmeyer, 2003)):

\[
\text{Description} ::= \ell: p(E_1, \ldots, E_n) | \neg\ell: p(E_1, \ldots, E_n) | E_i \ll E_j \quad (2)
\]

where \(\ell: p(E_1, \ldots, E_n)\) represents the predicate \(p\) with parameters \(E_1, \ldots, E_n\), and labeled \(\ell\). \(\neg\) is the logical negation, and \(E_i \ll E_j\) is the scope between \(E_i\) and \(E_j\) (used to deal with quantifiers).

Thus, one can write classes whose content consists of tree description and/or of semantic formulas. The XMG formalism furthermore supports the sharing of identifiers across dimension hence allowing for a straightforward encoding of the syntax/semantics interface (see figure 1).

3 Compiling a MetaGrammar into a Grammar

We now focus on the compilation process and on the constraint logic programming techniques we
As we have seen, an XMG metagrammar consists of classes that are combined. Provided these classes can be referred to by means of names, we can view a class as a clause associating a name with a content or goal to borrow vocabulary from Logic Programming. In XMG, this goal will be either a tree description, a semantic description, a name (class call) or a combination of classes (conjunction or disjunction). Finally, the valuation of a specific class can be seen as being triggered by a query.

\[
\text{Clause} ::= \text{Name} \rightarrow \text{Goal} \quad (3)
\]

\[
\text{Goal} ::= \text{Description} \mid \text{Name} \mid \text{Goal} \lor \text{Goal} \mid \text{Goal} \land \text{Goal} \quad (4)
\]

\[
\text{Query} ::= \text{Name} \quad (5)
\]

In other words, we view our metagrammar language as a specific kind of Logic Program namely, a Definite Clause Grammar (or DCG). In this DCG, the terminal symbols are descriptions.

To extend the approach to the representation of semantic information as introduced in 2, clause (4) is modified as follows:

\[
\text{Goal} ::= \text{Dimension} + \text{=Description} \mid \text{Name} \mid \text{Goal} \lor \text{Goal} \mid \text{Goal} \land \text{Goal}
\]

Note that, with this modification, the XMG language no longer correspond to a Definite Clause Grammar but to an Extended Definite Clause Grammar (see (Van Roy, 1990)) where the symbol \(\oplus\) represents the accumulation of information for each dimension.

**Virtual Machine** The evaluation of a query is done by a specific Virtual Machine inspired by the Warren’s Abstract Machine (see (Ait-Kaci, 1991)). First, it computes the derivations contained in the description, i.e. in the Extended Definite Clause Grammar, and secondly it performs unification of non standard data-types (nodes, node features for TAG). Eventually it produces as an output a description, more precisely one description per dimension (syntax, semantics).

In the case of TAG, the virtual machine produces a tree description. We still need to solve this description in order to obtain trees (i.e. the items of the resulting grammar).

**Constraint-based tree description solver** The tree description solver we use is inspired by (Duchier and Niehren, 2000). The idea is to:

1. associate to each node \(x\) in the description an integer,
2. then refer to \(x\) by means of the tuple \((\text{Eq}_x, \text{Up}_x, \text{Down}_x, \text{Left}_x, \text{Right}_x)\) where \(\text{Eq}_x\) (respectively \(\text{Up}_x, \text{Down}_x, \text{Left}_x, \text{Right}_x\)) denotes the set of nodes in the description which are equal, (respectively above, below, left, and right) of \(x\) (see picture 2). Note that these sets are set of integers.

\[
N^i \rightarrow N^j \equiv [N^i_{\text{EqUp}} \subseteq N^j_{\text{Up}} \land N^i_{\text{Down}} \supseteq N^j_{\text{EqDown}} \land N^i_{\text{Left}} \subseteq N^j_{\text{Left}} \land N^i_{\text{Right}} \subseteq N^j_{\text{Right}}]
\]

This means that if \(x\) dominates \(y\), then in a model, (1) the set of integers representing nodes that are equal or above \(x\) is included in the set of integers representing nodes that are strictly above \(y\).

\(N^i_{\text{EqUp}}\) corresponds to the disjoint union of \(N^i_{\text{Eq}}\) and \(N^i_{\text{Up}}\), similarly for \(N^i_{\text{EqDown}}\) with \(N^i_{\text{Eq}}\) and \(N^i_{\text{Down}}\).
the dual holds, i.e. the set of integers representing nodes that are below $x$ contains the set of integers representing nodes that are equal or below $y$, (3) the set of integers representing nodes that are on the left of $x$ is included in the set of integers representing those on the left of $y$, and (4) symmetrically for the nodes on the right.  

Parameterized constraint solver To recap from a grammar-designer's point of view, a queried class needs not define complete trees but rather a set of tree descriptions. The solver is then called to generate all the matching valid minimal trees from those descriptions. This feature provides the users with a way to concentrate on what is relevant in the grammar, thus taking advantage of underspecification, and to delegate the tiresome work to the solver.

Actually, the solver can be parameterized to perform various checks or constraints on the tree descriptions besides tree-shaping them. These parameters are called principles in the XMG terminology. Some are specific to a target formalism (e.g. TAG trees must have at most one foot node) while others are independent. The most interesting one is a resources/needs mechanism for node unification called color principle, see (Crabbé and Duchier, 2004).

At the end of this tree description solving process we obtain the trees of the grammar. Note that the use of constraint programming techniques to solve tree descriptions allows us to compute grammars faster than the previous approaches (see section 4).

4 Evaluation

The XMG system has been successfully used by linguists to develop a core TAG for French containing more than 6,000 trees. This grammar has been evaluated on the TSNLP test-suite, with a coverage rate of 75% (see (Crabbé, 2005)). The metagrammar used to produce that grammar consists of 290 classes and is compiled by the XMG system in about 16 minutes with a Pentium 4, 2.6 GHz and 1 GB of RAM.

XMG has also been used to produce a core size Interaction Grammar for French (see (Perrier, 2003)).

Finally, XMG is currently used to develop a TAG that includes a semantic dimension along the line described in (Gardent and Kallmeyer, 2003).

5 Conclusion and Future Work

We have presented a system, XMG\textsuperscript{8}, for producing broad-coverage grammars, system that offers an expressive description language along with an efficient compiler taking advantages from logic and constraint programming techniques.

Besides, we aim at extending XMG to a generic tool. That is to say, we now would like to obtain a compiler which would propose a library of languages (each associated with a specific processing) that the user would load dynamically according to his/her target formalism (not only tree-based formalisms, but others such as HPSG or LFG).

References

H. Ait-Kaci. 1991. Warren’s abstract machine: A tutorial reconstruction. In Proc. of the Eighth International Conference of Logic Programming.

T. Becker. 2000. Patterns in metarules. In A. Abeille and O. Rambow, editors, Tree Adjoining Grammars: formal, computational and linguistic aspects. CSLI publications, Stanford.

M.H. Candito. 1999. Représentation modulaire et paramétrable de grammaires électroniques lexicalisées : application au français et à l’italien. Ph.D. thesis, Université Paris 7.

B. Crabbé and D. Duchier. 2004. Metagrammar redux. In CSLP 2004, Copenhagen.

B. Crabbé. 2005. Représentation informatique de grammaires fortement lexicalisées : Application à la grammaire d’arbres adjoints. Ph.D. thesis, Université Nancy 2.

D. Duchier and J. Niehren. 2000. Dominance constraints with set operators. In Proceedings of CL2000.

B. Gaiffe, B. Crabbé, and A. Roussanaly. 2002. A new metagrammar compiler. In Proceedings of TAG+6.

C. Gardent and L. Kallmeyer. 2003. Semantic construction in ftag. In Proceedings of EACL’03.

Guy Perrier. 2003. Les grammaires d’interaction. HDR en informatique, Université Nancy 2.

P. Van Roy. 1990. Extended dcl notation: A tool for applicative programming in prolog. Technical report, Technical Report UCB/CSD 90/583, Computer Science Division, UC Berkeley.

\textsuperscript{8}XMG is freely available at \url{http://sourcesup.cru.fr/xmg}.