

Cross-framework Grammar Engineering using Constraint-driven Metagrammars

Denys Duchier, Yannick Parmentier, Simon Petitjean

▶ **To cite this version:**

Denys Duchier, Yannick Parmentier, Simon Petitjean. Cross-framework Grammar Engineering using Constraint-driven Metagrammars. 6th International Workshop on Constraint Solving and Language Processing (CSLP'11), Sep 2011, Karlsruhe, Germany. pp.32-43. hal-00614661

HAL Id: hal-00614661
https://hal.science/hal-00614661

Submitted on 19 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cross-framework Grammar Engineering using Constraint-driven Metagrammars

Denys Duchier, Yannick Parmentier, and Simon Petitjean

LIFO, Université d'Orléans, F-45067 Orléans Cedex 2, France,
firstname.lastname@univ-orleans.fr,
WWW home page: http://www.univ-orleans.fr/lifo/
– Preprint –

Abstract. In this paper, we present an abstract constraint-driven formalism for grammar engineering called *eXtensible MetaGrammar* and show how to extend it to deal with cross-framework grammar engineering. As a case study, we focus on the design of tree-adjoining, lexical-functional, and property grammars (TAG / LFG / PG).
A particularly interesting feature of this formalism is that it allows to apply specific constraints on the linguistic structures being described.

Keywords: computational linguistics, formal grammar, metagrammar, constraint solving.

1 Introduction

Many grammatical frameworks have been proposed over the last decades to describe the syntax of natural language. Among the most widely used, one may cite Tree-Adjoining Grammar (TAG) [1], Lexical-Functional Grammar (LFG) [2], or Head-driven Phrase Structure Grammar (HPSG) [3]. These frameworks present theoretical and practical interests. From a theoretical point of view, they provide a formal device for the linguist to experiment with her/his theories. From a practical point of view, they make it possible to automatically process natural language in applications such as dialog systems, machine translation, *etc.* They differ in their expressivity and complexity. Some reveal themselves more adequate for the description of a given language than others. Still, for many of these frameworks, large resources (*i.e.*, grammars) have been designed, at first by hand, and later via dedicated tools (*e.g.*, integrated grammar environments such as XLE for LFG [4]). In this paper, we are concerned with this complex task of grammar engineering, keeping in mind the two above-mentioned theoretical and practical interests.

Several approaches have been proposed for a computer-aided grammar engineering, mainly to reduce the costs of grammar extension and maintenance. The main approaches are 1. the automatic acquisition from treebanks (see *e.g.*, [5] for LFG), 2. systems based on an abstract description of the grammar, either via transformation rules, also known as *metarules* (see *e.g*, [6] for TAG) or via a description language, sometimes called *metagrammar* (see *e.g.*, [7] for TAG). The

advantage of the description-based approach (and especially metagrammars[1]) over the automatic acquisition approach lies in the linguistic control it provides. Indeed, these descriptions capture linguistic generalizations and make it possible to reason about language at an abstract level. Describing language at an abstract level is not only interesting for structure sharing within a given framework, but also for information sharing between frameworks and / or languages.

This observation was already made by [9,10]. In their papers, the authors showed how to extend an existing metagrammar for TAG so that both a TAG and an LFG could be generated from it. They annotated TAG metagrammatical elementary units (so-called classes) with extra pieces of information, namely (i) LFG's functional descriptions and (ii) filtering information to distinguish common classes from classes specific to TAG or LFG. The metagrammar compilation then generated an extended TAG, from which LFG rules were extracted. To maximize the structure sharing between their TAG and LFG metagrammars, the authors defined classes containing tree fragments of depth one. These fragments were either combined to produce TAG trees or associated with functional descriptions to produce LFG rules. This cross-framework experiment was applied to the design of a French / English parallel metagrammar, producing both a TAG and a LFG. This work was still preliminary. Indeed (i) it concerned a limited metagrammar (the target TAG was composed of 550 trees, and the associated LFG of 140 rules) (ii) more importantly, there is no clear evidence whether a generalization to other frameworks and / or languages could be possible (metagrammar implementation choices, such as tree fragment depth, were not independent from the target frameworks).

Here, we chose to adopt a more generalized approach by designing an extensible metagrammatical language, that can handle an arbitrary number of distinct target frameworks. The linguist can thus use the same formalism to describe different frameworks and grammars. Nonetheless, if one wants to experiment with multi-formalism, *e.g.*, by designing a parallel TAG / LFG grammar, nothing prevents her/him from defining "universal" classes, which contain metagrammatical descriptions built on a common sublanguage. Rather than designing a new metagrammatical language from scratch, we propose to extend an existing formalism, namely *eXtensible MetaGrammar* (XMG) [11], which seems particularly adequate thanks to its modularity and extensibility.

The paper is organized as follows. In section 2, we briefly introduce TAG, as well as the redundancy issues raising while developing large TAG grammars (which motivated metagrammars). We then introduce the XMG metagrammatical language and show how it can be used to design TAG grammars. In section 3, we briefly introduce LFG and present an extension of XMG to describe LFG grammars. In section 4, we introduce Property Grammar (PG) [12], and present a second extension of XMG to generate PG grammars. In section 5, we will generalize over these two extensions, and define a layout for cross-framework grammar engineering. Finally, we conclude and give perspectives in section 6.

[1] In rule-based descriptions, one has to carefully define the ordering of the applications of rules [8], which makes it hard to design large grammars.

2 eXtensible Meta Grammar: generating Tree-Adjoining Grammars with a metagrammar

2.1 Tree-Adjoining Grammar

TAG[2] is a tree rewriting system, where elementary trees can be combined via two rewriting operations, namely *substitution* and *adjunction*. Substitution consists in replacing a leaf node labelled with ↓ with a tree whose root has the same syntactic category as this leaf node. Adjunction consists in replacing an internal node with a tree where both the root node and one of the leaf nodes (labelled with ⋆) have the same syntactic category as this internal node. As an illustration, consider Fig. 1 below. It shows (i) the substitution of the elementary tree associated with the noun *John* into the elementary tree associated with the verb *sleeps*, and (ii) the adjunction of the elementary tree associated with the adverb *deeply* into the tree associated with *sleeps*.

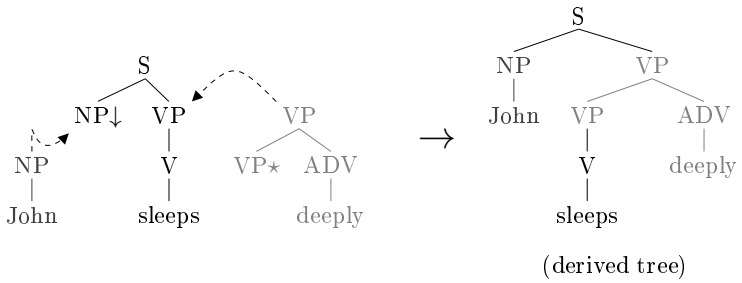


Fig. 1. Tree rewriting in TAG

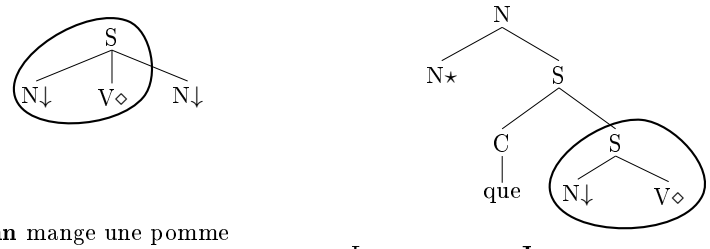
Basically, a real size TAG is made of thousands of elementary trees [14,15]. Due to TAG's extended domain of locality, many of these trees share common sub-trees, as for instance the relation between a canonical subject and its verb, as shown in Fig. 2. To deal with this redundancy, the metagrammar approach (in particular XMG) proposes to describe large TAG grammars in an abstract and factorized way.

2.2 eXtensible MetaGrammar (XMG)

XMG is a metagrammatical language inspired by logic programming. The idea behind XMG is that a metagrammar is a declarative logical specification of what a grammar is. This specification relies on the following three main concepts:

- several *dimensions* of language (*e.g.* syntax, semantics) can be described;
- for *each* of these dimensions, descriptions are made of non-deterministic combinations of elementary units;
- for *some* of these dimensions, descriptions must be solved to produce models.

[2] For a detailed introduction to TAG, see [13].



Jean mange une pomme
John eats an apple

La pomme que **Jean** mange
The apple that John eats

Fig. 2. Structural redundancy in TAG

XMG's extensibility comes from the concept of dimensions. These allow to describe an arbitrary number of types of linguistic structures. Non-determinism allows for factorization, and description solving for assembly and validation (*i.e.*, well-formedness of the description according to some target framework). Hereafter, we will first use XMG to describe TAG. Then, we will apply XMG's extensibility to the description of other frameworks, namely LFG and PG. Eventually, we will generalize over these applications.

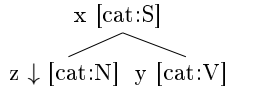
When describing TAG trees with XMG, one defines both (i) tree fragments and (ii) constraints that express how these fragments have to be combined to produce the grammar. Two languages are thus used: a *description language* $\mathcal{L}_D$ to specify fragments, and a *control language* $\mathcal{L}_C$ to specify combination constraints.

$\mathcal{L}_D$ is based on the precedence and dominance relations. Furthermore, since TAG allows for the labelling of syntactic nodes with feature structures, so does $\mathcal{L}_D$. A description in $\mathcal{L}_D$ is a formula built as follows:

$$Desc := x \rightarrow y \mid x \rightarrow^{+} y \mid x \rightarrow^{*} y \mid x \prec y \mid x \prec^{+} y \mid x[f{:}E] \mid x(p{:}E) \mid Desc \wedge Desc$$

where x, y refer to node variables, $\rightarrow$ (resp. $\prec$) to the dominance (resp. precedence) relation, and $+$ (resp. $*$) are used to denote the transitive (resp. reflexive and transitive) closure of this relation. The square brackets are used to associate a node variable with some feature structure. Parenthesis are used to associate a node variable with some property (such as the TAG $\star$ property seen in Fig. 1). Note that node variables are *by default* local to a description. If a node variable needs to be accessed from outside its description, it is possible to use some *export* mechanism. Once a variable is exported, it becomes accessible using a dot operator. For instance, to refer to the variable x in the description $Desc$, one writes $Desc.x$. Here is an illustration of a fragment description in XMG (on the right, one can see a minimal model of this description):

$$(x\ [cat : S] \rightarrow y\ [cat : V]\ ) \wedge (x \rightarrow z\ (mark : subst)\ [cat : N]\ ) \wedge (z \prec y)$$



$\mathcal{L}_C$ offers three mechanisms to handle fragments: *abstraction* via parameterized *classes* (association of a name and zero or more parameters with a content),

conjunction (accumulation of contents), and *disjunction* (non-deterministic accumulation of contents). A formula in $\mathcal{L}_C$ is built as follows:

$$\begin{array}{rcl} \mathit{Class} & := & \mathit{Name}[p_1, \ldots, p_n] \rightarrow \mathit{Content} \\ \mathit{Content} & := & \mathit{Desc} \mid \mathit{Name}[\ldots] \mid \mathit{Content} \vee \mathit{Content} \mid \mathit{Content} \wedge \mathit{Content} \end{array}$$

As an illustration of $\mathcal{L}_C$, let us consider different object realizations. One could for instance define the 4 fragments: (i) canonical subject, (ii) verbal morphology, (iii) canonical, and (iv) relativized object, and the following combinations, thus producing the two trees of Fig. 2:

$$\begin{array}{c} \mathit{Object} \rightarrow \mathit{CanObj} \vee \mathit{RelObj} \\ \mathit{Transitive} \rightarrow \mathit{CanSubj} \wedge \mathit{VerbMorph} \wedge \mathit{Object} \end{array}$$

Metagrammar compilation. To produce a grammar from an XMG metagrammar, we let the logical specification generate, in a non-deterministic way, descriptions. In other words, the combination constraints are processed to generate descriptions (one per dimension). For some dimensions, descriptions need to be solved to produce models. This is the case for TAG, a constraint-based tree description solver is thus used to compute trees [11]. Note this solver actually checks several types of constraints [16]: tree well-formedness constraints, TAG-related constraints (*e.g.*, unique node labelled $\star$), and language-related constraints (*e.g.*, uniqueness and order of clitics in French).

As one of the first ambitions of XMG is multi-formalism, dimensions are an efficient way to define different types of description language adapted to target frameworks. Let us see how to define dimensions for LFG and PG.

3 Generating Lexical-Functional Grammars with a metagrammar

3.1 Lexical-Functional Grammar

A lexical-functional grammar (LFG) consists of three main components: 1. context-free rules annotated with functional descriptions, 2. well-formedness principles, and 3. a lexicon. From these components, two main interconnected structures can be built[3]: a c(onstituent)-structure, and a f(unctional)-structure. The c-structure represents a syntactic tree, and the f-structure grammatical functions in the form of recursive attribute-value matrices. As an example of LFG, consider the Fig. 3 below. It contains a toy grammar and the c- and f-structures for the sentence "John loves Mary". In this example, one can see functional descriptions labelling context-free rules (see (1) and (2)). These descriptions are made of equations. For instance, in rule (1), the equation $(\uparrow SUBJ) = \downarrow$ constrains the $SUBJ$ feature of the functional description associated with the left-hand side of the context-free rule to *unify* with the functional description associated with

[3] This connection is often referred to as *functional projection* or *functional mapping*.

the first element of the right-hand side of the rule. In other words, these equations are unification constraints between attribute-value matrices. Nonetheless, these constraints may not provide enough control on the f-structures licensed by the grammar, LFG hence comes with three additional well-formedness principles (completeness, coherence and uniqueness) [2].

Toy grammar:

(1) S → NP VP
$\uparrow=\downarrow$ $(\uparrow SUBJ)=\downarrow$ $\uparrow=\downarrow$

(2) VP → V NP
$\uparrow=\downarrow$ $\uparrow=\downarrow$ $(\uparrow OBJ)=\downarrow$

(3) John NP, $(\uparrow PRED) =' JOHN', (\uparrow NUM) = SG, (\uparrow PRES) = 3$

(4) Mary NP, $(\uparrow PRED) =' MARY', (\uparrow NUM) = SG, (\uparrow PRES) = 3$

(5) loves V, $(\uparrow PRED) =' LOVE\langle(\uparrow SUBJ)\,(\uparrow OBJ)\rangle', (\uparrow TENSE) = PRESENT$

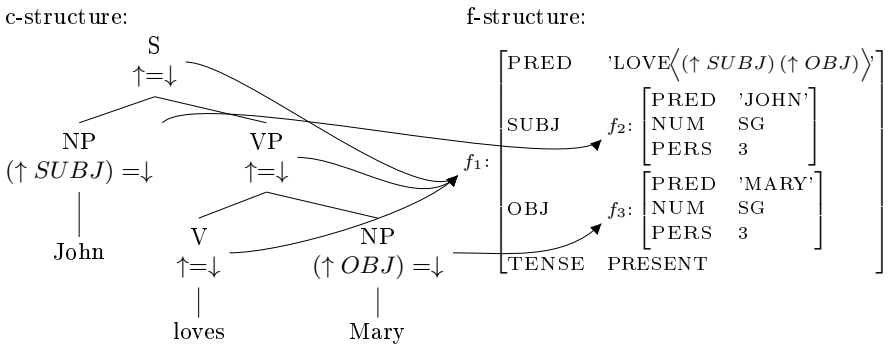


Fig. 3. LFG grammar and c-and f-structures for the sentence "John loves Mary"

3.2 Extending XMG for LFG

In the previous section, we defined the XMG language, and applied it to the description of TAG. Let us recall that one of the motivations of metagrammars in general (and of XMG in particular) is the redundancy which affects grammar extension and maintenance. In TAG, the redundancy is higher than in LFG. Still, as mentioned in [9], in LFG there are redundancies at different levels, namely within the rewriting rules, the functional equations and the lexicon. Thus, the metagrammar approach can prove helpful in this context. Let us now see what type of language could be used to describe LFG.[4]

To describe LFG at an abstract level, one needs to describe its elementary units, which are context-free rules annotated with functional descriptions (*e.g.*, equations) and lexical entries using attribute-value matrices. Context-free rules

[4] A specification language for LFG has been proposed by [17], but it corresponds more to a model-theoretic description of LFG than to a metagrammar.

can be seen as trees of depth one. Describing such structures can be done in XMG using a description language similar to the one for TAG, *i.e.*, using the $\rightarrow$ (dominance) and $\prec$ (precedence) relations. One can for instance define different context-free backbones according to the number of elements in the right-hand sides of the LFG rules. These backbones are encapsulated in parameterized XMG classes, where the parameters are used to assign a syntactic category to a given element of the context-free rule, such as in the class *BinaryRule* below.

$$\begin{array}{l} \mathit{BinaryRule}[A,B,C] \;\; \rightarrow \;\; (x[cat:A] \rightarrow y[cat:B]) \wedge (x \rightarrow z[cat:C]) \wedge (y \prec^{+} z) \\ \quad \mathit{exports} \;\; \langle x,y,z \rangle \end{array}$$

We also need to annotate the node variables x, y, z with functional descriptions. Let us see how these functional descriptions F_{Desc} are built:[5]

$$\begin{array}{rl} F_{desc} := & \exists(g\ FEAT) \mid \neg(g\ FEAT) \mid\ (g * FEAT) \mid\ (g\ FEAT)\ \ CONST\ \ VAL \mid \\ & F_{desc} \vee F_{desc}\ \mid\ (F_{desc})\ \mid\ F_{desc} \wedge F_{desc} \end{array}$$

where g refers to an attribute-value matrix, $FEAT$ to a feature, VAL to a (possibly complex) value, $CONST$ to a constraint operator ($=$ for unification, $=_c$ for constraining unification, $\in$ for set membership, $\neq$ for difference), (F_{Desc}) to optionality, and $*$ to LFG's functional uncertainty. Note that g can be complex, that is, it can correspond to a (relative – using $\uparrow$ and $\downarrow$ – or absolute) path pointing to a sub-attribute-value matrix.

To specify such functional descriptions, we can extend XMG in a straightforward manner, with a dedicated dimension and a dedicated description language $\mathcal{L}_{LFG}$ defined as follows:

$$\begin{array}{rcl} \mathit{Desc}_{LFG} & := & x \rightarrow y \ \mid\ x \prec y \ \mid\ x \prec^{+} y \ \mid\ x = y \ \mid\ x[f{:}E] \ \mid\ x\langle F_d \rangle \ \mid \\ & & \mathit{Desc}_{LFG} \ \wedge \ \mathit{Desc}_{LFG} \\ F_d & := & g \mid\ \exists g.f \mid\ g.f = v \mid\ g.f =_c v \mid\ g.f \in v \mid\ \neg F_d \mid\ F_d \vee F_d \mid \\ & & (F_d) \mid\ F_d \wedge F_d \\ g,h & := & \uparrow \mid \downarrow \mid\ h.f \mid\ f * i \end{array}$$

where g, h are variables denoting attribute-value matrices, f, i (atomic) feature names, v (possibly complex) values, and $\langle \ldots \rangle$ corresponds to LFG's functional mapping introduced above. With such a language, it now becomes possible to define an XMG metagrammar for our toy LFG as follows.[6]

$$\begin{array}{rl} \mathit{Srule} & \rightarrow br = \mathit{BinaryRule}[\texttt{S},\texttt{NP},\texttt{VP}] \wedge br.x\langle\uparrow{=}\downarrow\rangle \wedge br.y\langle(\uparrow \texttt{.SUBJ}) {=}\downarrow\rangle \\ & \quad \wedge\ br.z\langle\uparrow{=}\downarrow\rangle \\ \mathit{VPrule} & \rightarrow br = \mathit{BinaryRule}[\texttt{VP},\texttt{V},\texttt{NP}] \wedge br.x\langle\uparrow{=}\downarrow\rangle \wedge br.y\langle\uparrow{=}\downarrow\rangle \\ & \quad \wedge\ br.z\langle(\uparrow \texttt{.OBJ}) {=}\downarrow\rangle \end{array}$$

[5] We do not consider here additional LFG operators, which have been introduced in specific LFG environments, such as *shuffle*, *insert or ignore*, *etc.*

[6] Here, we do not describe the lexical entries, these can be defined using the same language as the LFG context-free rules, omitting the right-and-side.

In this toy example, the structure sharing is minimal. To illustrate what can be done, let us have a look at a slightly more complex example taken from [9]:

$$\begin{array}{lcccc} VP \rightarrow & V & (NP) & PP & (NP) \\ & \uparrow{=}\downarrow & (\uparrow OBJ) {=}\downarrow & (\uparrow SecondOBJ) {=}\downarrow & (\uparrow OBJ) {=}\downarrow \end{array}$$

Here, we have two possible positions for the NP node, either before or after the PP node. Such an situation can be described in XMG as follows:

$$\begin{array}{l} VPrule2 \rightarrow br = BinaryRule[\texttt{VP}, \texttt{V}, \texttt{PP}] \ \wedge \ u[cat : \texttt{NP}] \ \wedge \ br.y \prec^{+} u \\ \qquad\qquad \wedge \ br.y\langle\uparrow{=}\downarrow\rangle \ \wedge \ br.z\langle(\uparrow \texttt{.SecondOBJ}) {=}\downarrow\rangle \ \wedge \ u\langle(\uparrow \texttt{.OBJ}) {=}\downarrow\rangle \end{array}$$

Here, we do not specify the precedence between the NP and PP nodes. We simply specify that the NP node is preceded by the V node (denoted by y). When compiling this description with a solver such as the one for TAG, two solutions (LFG rules) will be computed. In other terms, the optionality can be expressed directly at the metagrammatical level, and the metagrammar compiler can directly apply LFG's uniqueness principle.

In other words, the metagrammar here not only allows for structure sharing via the (conjunctive or disjunctive) combination of parameterized classes, but it also allows to apply well-formedness principles to the described structures. In the example above with the two NP nodes, this well-formedness principle is checked on the constituent structure and indirectly impacts the functional structure (which is the structure concerned with these principles). If we see the functional structures as graphs and equations as constraints on these, one could imagine to develop a specific constraint solver. This would allow to turn the metagrammar compiler into an LFG parser, which would, while solving tree descriptions for the constituent structure, solve graph-labelling constraints for the functional structure.

Note that a similar approach of structure sharing within an LFG through combinations of elementary units has been proposed by [18]. In their paper, the authors describe how to share information between LFG structures by defining named descriptions, called templates. These templates can abstract over conjunction or disjunction of templates, they are thus comparable to our metagrammar classes. The main difference with our approach, is that nothing is said about an interpretation of these templates (they act in a macro-like fashion), while in XMG, one could apply some specific treatments (*e.g.* constraint solving) on the metagrammar classes.

4 Generating Property Grammars with a metagrammar

4.1 Property Grammar

Property Grammar (PG) [12] differs from TAG or LFG in so far as it does not belong to the generative syntax family, but to the model-theoretic syntax one. In PG, one defines the relations between syntactic constituents not in terms

of rewriting rules, but in terms of local constraints (the so-called *properties*).[7] The properties licensed by the framework rely on linguistic observations, such as linear precedence between constituents, coocurrency, mutual exclusion, *etc.*

Here, we will consider the following 6 properties, that constrain the relations between a constituent (*i.e.*, the node of a syntactic tree), with category A and its sub-constituents (*i.e.*, the daughter-nodes of A):[8]

Obligation	$A : \triangle B$	at least one B child
Uniqueness	$A : B!$	at most one B child
Linearity	$A : B \prec C$	B child precedes C child
Requirement	$A : B \Rightarrow C$	if a B child, then also a C child
Exclusion	$A : B \not\Rightarrow C$	B and C children are mutually exclusive
Constituency	$A : S$	children must have categories in S

In a real size PG, such as the French PG of [19], these properties are encapsulated (together with some syntactic features) within *linguistic constructions*, and the latter arranged in an inheritance hierarchy[9]. An extract of the hierarchy of [19] is presented in Fig. 4 (fragment corresponding to basic verbal constructions).

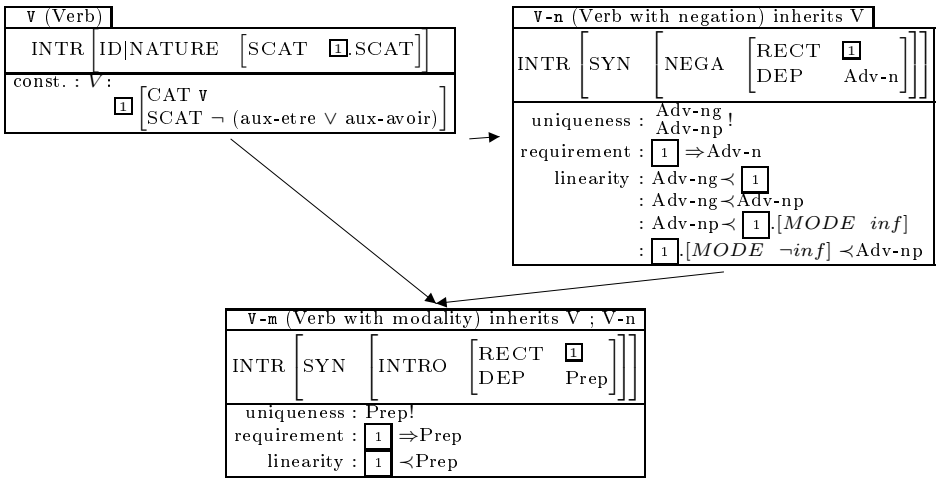


Fig. 4. Fragment of a PG for French (basic verbal constructions)

Let us for instance have a closer look at the properties of the `V-n` construction of Fig. 4. It says that in French, for verbs with a negation, this negation is made

[7] An interesting characteristic of these constraints is that they can be independently violated, and thus provide a way to characterize agrammatical sentences.

[8] Here, we omit lexical properties, such as $\mathsf{cat}(apple) = \mathtt{N}$.

[9] Note that this hierarchy is a disjunctive inheritance hierarchy, *i.e.*, when there is multiple inheritance, the subclass inherits *one* of its super-classes.

of an adverb *ne* (labelled with the category *Adv-ng*) and / or an adverb *pas* (or a related adverb such as *guère*, labelled with the category *Adv-np*). These adverbs, if they exist, are unique (*uniqueness* property), and linearly ordered (*linearity* property). When the verb is an infinitive, it comes after these adverbs (*e.g., ne pas donner (not to give)* versus *je ne donne pas (I do not give)*).

4.2 Extending XMG for PG

In order to describe PG, we need to extend the XMG formalism with linguistic constructions. These will be encapsulated within XMG's classes. As for LFG, we extend XMG with a dedicated dimension and a dedicated description language $\mathcal{L}_{PG}$. Formulas in $\mathcal{L}_{PG}$ are built as follows:

$$\begin{aligned} Desc_{PG} &:= x \mid x = y \mid x \neq y \mid [f{:}E] \mid \{P\} \mid Desc_{PG} \wedge Desc_{PG} \\ P &:= A : \triangle B \mid A : B! \mid A : B \prec C \mid A : B \Rightarrow C \mid A : B \not\Leftrightarrow C \mid A : B \end{aligned}$$

where x, y correspond to unification variables, $=$ to unification, $\neq$ to unification failure, E to some (possibly complex) expression to be associated with the feature f, and $\{P\}$ to a set of properties. Note that E and P may share unification variables. With this language, it is now possible to define the above `V`, `V-n` and `V-m` constructions as follows:

$$\begin{aligned} Vclass \rightarrow\ & [\mathsf{INTR} : [\mathsf{ID}|\mathsf{NATURE} : [\mathsf{CAT} : X.\mathsf{SCAT}]]] \wedge (V : X) \\ & \wedge (X = [\mathsf{CAT} : V, \mathsf{SCAT} : Y]) \wedge (Y \neq \texttt{aux-etre}) \wedge (Y \neq \texttt{aux-avoir}) \\ V{-}n \rightarrow\ & Vclass \wedge [\mathsf{INTR}{:}[\mathsf{SYN}{:}[\mathsf{NEGA}{:}[\mathsf{RECT}{:}X, \mathsf{DEP}{:}\texttt{Adv-n}]]]] \\ & \wedge (V : \texttt{Adv-ng!}) \wedge (V : \texttt{Adv-np!}) \wedge (V : X \Rightarrow \texttt{Adv-n}) \\ & \wedge (V : \texttt{Adv-ng} \prec X) \wedge (V : \texttt{Adv-ng} \prec \texttt{Adv-np}) \\ & \wedge (V : \texttt{Adv-ng} \prec Y) \wedge (V : Z \prec \texttt{Adv-np}) \\ & \wedge (Y = \texttt{inf}) \wedge (Y = X.\mathsf{mode}) \wedge \neg(Z = \texttt{inf}) \wedge (Z = X.\mathsf{mode}) \\ V{-}m \rightarrow\ & (Vclass \vee V{-}n) \wedge [\mathsf{INTR}{:}[\mathsf{SYN}{:}[\mathsf{INTRO}{:}[\mathsf{RECT}{:}X, \mathsf{DEP}{:}\texttt{Prep}]]]] \\ & \wedge (V : \texttt{Prep!}) \wedge (V : X \Rightarrow \texttt{Prep}) \wedge (V : X \prec \texttt{Prep}) \end{aligned}$$

Note that the disjunction operator from XMG's control language $\mathcal{L}_C$ allows us to represent [19]'s disjunctive inheritance. Also, compared with TAG and LFG, there is relatively few redundancy in PG, for redundancy is already dealt with directly at the grammar level, by organizing the constructions within an inheritance hierarchy based on linguistic motivations.

As for LFG, the metagrammar could be extended so that it solves the properties contained in the final classes, according to a sentence to parse. This could be done by adding a specific constraint solver such as that of [20] as a post-processor of the metagrammar compilation.

5 Towards an extensible metagrammatical formalism

We have seen two extensions of the XMG formalism to describe not only TAG grammars, but also LFG and PG ones, these rely on the following concepts:

- The metagrammar describes a grammar by means of conjunctive and / or disjunctive combinations of elementary units (using a combination language $\mathcal{L}_C$).
- The elementary units of the (meta)grammar depend on the target framework, and are expressed using dedicated description languages ($\mathcal{L}_D$,$\mathcal{L}_{LFG}$,$\mathcal{L}_{PG}$).

When compiling a metagrammar, the compiler executes the logic program underlying $\mathcal{L}_C$ (*i.e.*, unfolds the combination rules) while storing the elementary units of $\mathcal{L}_{D|LFG|PG}$ in dedicated accumulators. The resulting accumulated descriptions may need some additional post-processing (*e.g.*, tree description solving for TAG). Thus, to extend XMG into a cross-framework grammar engineering environment, one needs (i) to design dedicated description languages, and (ii) to develop the corresponding pre/post-processing modules (*e.g.*, metagrammar parsing / description solving).

A first version of XMG (XMG 1) was developed in Oz-Mozart.[10] It implements the language described in section 2, and supports tree-based formalisms, namely TAG and Interaction Grammar [21]. It has been used to design various large tree grammars for French, English and German.[11] The implementation of a new version of XMG (XMG 2) has started in 2010, in Prolog (with bindings to the Gecode Constraint Programming C++ library)[12], with the goal of supporting cross-framework grammar engineering as presented here.

6 Conclusion and perspectives

In this paper, we presented a metagrammatical formalism for cross-framework grammar engineering. This formalism offers a collection of description languages, making it possible to describe different types of linguistic structures (TAG's syntactic trees, LFG's functional descriptions, PG's linguistic constructions), these structures being combined either conjunctively or disjunctively via a common control language. The formalism also applies specific constraints on some of these structures to ensure their well-formedness (*e.g.*, rank principle for TAG).

Using a formalism that can describe several types of grammar frameworks offers new insights in grammar comparison and sharing. This sharing appears naturally when designing parallel grammars, but appears also when designing distinct grammars (*e.g.*, reuse of the combinations of elementary units).

The implementation of the formalism introduced here is a work in progress. We aim to provide the linguist with an extensible formalism, offering a rich collection of predefined description languages; each one with a library of principles, and constraint solvers to effect specific assembly, filtering, and verifications on the grammatical structures described by the metagrammar.

[10] See `http://sourcesup.cru.fr/xmg` and `http://www.mozart-oz.org`.

[11] These are available on line, see `http://sourcesup.cru.fr/projects/xmg` (repository METAGRAMMARS) and `http://www.sfs.uni-tuebingen.de/emmy/res-en.html`.

[12] See `https://launchpad.net/xmg` and `http://www.gecode.org`.

References

1. Joshi, A.K., Levy, L.S., Takahashi, M.: Tree adjunct grammars. Journal of the Computer and System Sciences **10** (1975) 136–163
2. Bresnan, J.: The passive in lexical theory. In Bresnan, J., ed.: The Mental Representation of Grammatical Relations. The MIT Press, Cambridge, MA (1982)
3. Pollard, C., Sag, I.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Stanford : CSLI Publications, Chicago (1994)
4. King, T.H., Dipper, S., Frank, A., Kuhn, J., Maxwell, J.: Ambiguity management in grammar writing. In: Proceedings of the Workshop on Linguistic Theory and Grammar Implementation, ESSLLI 2000, Birmingham, Great-Britain (2000)
5. Cahill, A.: Parsing with Automatically Acquired, Wide-Coverage, Robust, Probabilistic LFG Approximations. PhD thesis, Dublin City University (2004)
6. Becker, T.: Patterns in Metarules for TAG. In: Tree Adjoining Grammars. Formalisms, Linguistic Analysis and Processing. CSLI, Stanford (2000)
7. Candito, M.: A Principle-Based Hierarchical Representation of LTAGs. In: Proceedings of COLING 96, Copenhagen, Denmark (1996)
8. Prolo, C.: Systematic grammar development in the XTAG project. In: Proceedings of COLING'02, Taipei, Taiwan (2002)
9. Clément, L., Kinyon, A.: Generating parallel multilingual LFG-TAG grammars from a MetaGrammar. In: Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL), Sapporo, Japan (2003)
10. Clément, L., Kinyon, A.: Generating LFGs with a MetaGrammar. In: Proceedings of LFG-03, Saratoga Springs, United States of America (2003)
11. Duchier, D., Le Roux, J., Parmentier, Y.: The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture. In: Proceedings of the 2nd Oz-Mozart Conference, MOZ 2004, Charleroi, Belgium (2004)
12. Blache, P.: Constraints, Linguistic Theories and Natural Language Processing. Lecture Notes in Artificial Intelligence Vol. 1835. Springer-Verlag (2000)
13. Joshi, A.K., Schabès, Y.: Tree adjoining grammars. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Springer Verlag, Berlin (1997)
14. XTAG Research Group: A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania (2001)
15. Crabbé, B.: Représentation informatique de grammaires fortement lexicalisées : Application à la grammaire d'arbres adjoints. PhD thesis, Université Nancy 2 (2005)
16. Le Roux, J., Crabbé, B., Parmentier, Y.: A constraint driven metagrammar. In: The Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+8), Sydney, Australia (2006)
17. Blackburn, P., Gardent, C.: A Specification Language for Lexical Functional Grammars. In: Proceedings of EACL'95, Dublin, Ireland (1995)
18. Dalrymple, M., Kaplan, R., King, T.H.: Lexical structures as generalizations over descriptions. In: Proceedings of LFG 04, Christchurch, New Zealand (2004)
19. Guénot, M.L.: Éléments de grammaire du français pour une théorie descriptive et formelle de la langue. PhD thesis, Université de Provence (2006)
20. Duchier, D., Dao, T.B.H., Parmentier, Y., Lesaint, W.: Property Grammar Parsing Seen as a Constraint Optimization Problem. In: Proceedings of the 15th International Conference on Formal Grammar (FG 2010), Copenhagen, Denmark (2010)
21. Perrier, G.: Interaction Grammars. In: Proceedings of COLING 2000, Saarbrücken, Germany (2000)