XMG/Documentation

Un article de Loria Wiki.

< XMG

If you want to know "how to produce grammars using XMG" this documentation is for you.
Current XMG version: 1.1.6.

Sommaire

[masquer]

1 Introduction

The XMG system corresponds to what is usually called a "metagrammar compiler" (see below). More precisely it is a tool for designing large scaled grammars for natural language. Provided a compact representation of grammatical information, XMG combines elementary fragments of information to produce a fully redundant strongly lexicalised grammar. It is worth noticing that by XMG, we refer to both

- a formalism allowing one to describe the linguistic information contained in a grammar,
- a device computing grammar rules from a description based on this formalism.

1.1 What is a metagrammar ?

This term has been introduced at the end of the 1990s by MH Candito. During her PhD, she proposed a new process to generate semi-automatically a Tree Adjoining Grammar (TAG) from a reduced description that captures the linguistic generalizations appearing among the trees of the grammar. This reduced description is the `metagrammar`.

1.2 What is a metagrammar compiler ?

Once we have described the grammar rules by specifying the way structure is shared, i.e. by defining reusable fragments, we use a specific tool to combine these. Such a tool is called a `metagrammar compiler`.

1.3 A brief introduction to XMG

XMG stands for eXtensible MetaGrammar. This system has been designed and implemented starting from springs 2003 in the framework of Benoit Crabbé's PhD. This project has been led by Dr Denys Duchier. It has also involved (and still involves) several other people, in particular two PhD students: Joseph Le Roux and Yannick Parmentier, and their supervisors: Guy Perrier and Claire Gardent. This system is freely available under the terms of the CeCILL license. XMG's web page is http://sourcesup.cru.fr/xmg.

1.4 Using this documentation

This documentation aims at helping people to use the XMG system. We advise you to have a look at the table of content and access directly the chapters that are pertinent according to your background. Very often this documentation will be the starting point for writing metagrammars. To do so, you will find the description of XMG's concrete syntax in the section Designing a metagrammar .

2 A first session with the XMG system

In this section, we will see how to set up XMG's environment, and also start playing with a toy-metagrammar.

2.1 Getting XMG : requirements

- First of all, XMG has been developed in Oz using the Mozart programming system. Thus you need to install the Oz compiler (version >= 1.3.1), which is freely available at http://www.mozart-oz.org/download/view.cgi for the Linux, Windows and Macintosh platforms.

N.B.

  - Notice that under MacOS, you need to install the MacOS SDK in order to have a C++ compiler (see http://developer.apple.com/sdk/index.html). Similarly, under Windows, you need to have access to a GNU c++ compiler (for instance by using cygwin).
  - Besides, on Debian systems, the packages needed are mozart and mozart-stdlib, optionally you can also install mozart-gtk and mozart-doc.

- Once Oz-Mozart has been installed, you need to install the `Select` library available at http://www.mozart-oz.org/mogul/info/duchier/select.html. The installation is performed by invoking the following command:

```
ozmake --install --package=duchier-select-xxx.pkg
```

- Eventually, you are ready to download XMG and install it. You may either download the last version of XMG's sources through Subversion repository with anonymous access:

```
svn checkout http://subversion.cru.fr/xmg/ local_repository
```

or download it as an ozmake package at https://sourcesup.cru.fr/frs/?group_id=99. Note that since the version 1.3.1 of Mozart, `ozmake` is part of the system, so you don't need to install any additional package once Oz-Mozart is installed to use the `ozmake` command.

We recommend to use the sources as packages do not always contain the last updates (use the package when it is very recent).

2.2 Installing XMG

- If you downloaded XMG's source, you can install XMG by going into the source's directory and typing:

```
ozmake --upgrade
```

- If you downloaded XMG's ozmake package, invoke:

```
ozmake --install --package=xmg-xxx.pkg
```

If this package was already installed on your computer and you want to upgrade to a newer version, invoke:

```
ozmake --upgrade --package=xmg-xxx.pkg
```

or else ozmake will complain.

By default ozmake installs all binary files in the user's `~/.oz/1.3.1/bin` directory tree.

NB: in all cases, don't forget to update your `$PATH` environment variable so that XMG's binary files are available in this path. It will allow you to call XMG from anywhere, by invoking:

```
MetaTAG metagrammar_file.mg
```

2.3 Compiling a toy-metagrammar

The XMG system includes a toy metagrammar that we highly recommend to manipulate. If you installed XMG by means of the ozmake package (*i.e.* you don't have this metagrammar), you can download it at the following address: http://sourcesup.cru.fr/xmg/TagExample.mg. To compile it, just type:

```
MetaTAG TagExample.mg
```

(see also List of XMG's options below)
as a result, you will obtain the following graphical window:

1. First window: valuations
   Fenêtre 1

   This window will display the list of the tree families. A family corresponds to a `valuation`. In this case it means that, at the end of the metagrammar, there is the following specific instruction:

   ```
   value lexemeManger
   ```

   That is to say that the fragment called `lexemeManger` contains the information needed to build 0, 1 or more TAG trees.

2. Second window: solutions
   Fenêtre 2
   Once you have chosen to browse all the trees belonging to this family by selecting `lexemeManger`, a second window appears. Here there is only one tree in the family.
3. Third window: the TAG tree
   Fenêtre 3
   The tree is finally displayed. Several pieces of information are associated with a tree, namely a trace, an interface, syntactic and semantic structures. These will be explained in XMG's output

NB: note that there is no input graphical user interface in XMG. Nonetheless, *vim* and *emacs* modes are available at XMG's web page so that you can relatively comfortably develop your metagrammar with these editors. A metagrammar corresponds to a text file, usually with a .mg extension (not mandatory).

In the next section, we will see in detail how to write a metagrammar, that is to say, we will have a close look at XMG's syntax. At the same time we will introduce the concepts underlying XMG.

3 Designing a metagrammar

Here, we will see in details how to write a metagrammar with XMG. First the ideas underlying the system will be introduced. Then we will show the kind of information that is contained in a metagrammar. After that we will implement a metagrammar describing a small Tree Adjoining Grammar (TAG). Next we will see how to extend the metagrammatical description to support several

levels of linguistic description. Finally we will dissect XMG's output.

3.1 XMG's underlying ideas

3.1.1 Metagrammars as logic programs

A fundamental idea in XMG's formalism is that a metagrammar can be seen as a logic program. Let us take the TAG example. In this syntactical formalism, the structures describing the syntax are trees. In wide grammars, there is a huge redundancy among the trees. The intuition is to build the trees starting from reusable tree fragments that will be combined. If we consider that a tree fragment is a *fact*, then we have *rules* to combine these fragments. At the end, we ask for a combination to take place by using a *query*. This is summarized in the following table:

Logic Programming	**XMG**
rule's head	name of a fragment
rule's body	call of a fragment, Combination of fragments
fact	tree fragment
query	valuation

More precisely, we will not handle tree fragments but descriptions of such fragments. For instance, let us consider the following tree fragment :

Tree fragment

It is represented by the formula: (S \rightarrow N) ~~ \wedge ~~ (S \rightarrow V) ~~ \wedge ~~ (N > V)

where \rightarrow stands for *domination* and > for *precedence*.
Back to our comparison between metagrammars and logic programs, we can refine it by stating that *clauses* correspond to the association between a `Name` and a `Goal`. This `Goal` can either be a `Description`, a `Call`, a `Conjunction` or a `Disjunction`:

Clause	::=	Name \rightarrow Goal	(1)
Goal	::=	Description \mid\ Name \mid\ Goal \vee Goal \mid\ Goal \wedge Goal	(2)
Query	::=	*Name*	(3)

At the end, we obtain a metagrammatical core language that corresponds in fact to a *Definite Clause Grammar* (see References).

Thus our metagrammatical formalism provides a representation language allowing one to:

- refer to a tree fragment by an **abstraction** (i.e. a *Name* associated to different kind of information)
- combine two tree fragments **conjunctively**
- combine two tree fragments **disjunctively**

The role of such a language is to allow us to solve two problems that arise while developping grammars:

1. reach a good *factorization* (in the structure sharing)
2. finely *control* the way fragments are combined.

3.1.2 Information reuse and specialization

There is another important point that has to be taken into account in this context: how do we manage the scope of the (node) names used in a tree fragment ? That is to say, how do we *reuse and specialize* a fragment ? In XMG, we have made the following choice : the **default scope of a (node) name is the `clause`**. Thus if we name a given node with the identifier *X*, then we can reuse *X* in another fragment without any risk of conflict. The question that arises next is how to reuse fragments with specialization since there is no name sharing ? To support it, XMG integrates an **`export / import`** process.

- First, for each fragment you specify what names are exported.

More precisely, this means that with each `clause` is associated a record of the exported names:

Clause ::= \langle I_1,\ldots,I_n \rangle \Leftarrow Name \rightarrow Goal

When this fragment is reused, one can access the I_i identifier by using a dot operator (see XMG's concrete syntax: TAG example).

- Second, you can import a fragment, then you will access all the names that have been exported by it.

In the section XMG's concrete syntax: TAG example, we will see how to concretely use this `export / import` process on an example.

3.1.3 Different levels of linguistic description

One important feature of the XMG system is that it deals not only with syntactic tree structures, but also with other linguistic levels of description. We call such levels **dimensions**. For instance, one may want to describe tree descriptions of an Interaction Grammar (IG) instead of TAG trees, or also produce a TAG augmented with semantics. This can be done by using *dimensions*, that will distinguish different information. Each dimension is equipped with its own representation language and combining process. Formally our core language is extended by replacing (2) by the following expression:

Goal ::= (Dimension += Description) \mid Name \mid Goal \vee Goal \mid Goal \wedge Goal

Thus we extend our formalism going from *Definite Clause Grammars* to *Extended Definite Clause Grammars* (see References). The handling of dimensions is further detailed in Extension to other descriptive levels (semantics, etc): handling dimensions.

3.1.4 Ensuring well-formedness of the grammar (TAG)

Historically, metagrammars have been invented for producing TAGs. This is why several features of metagrammatical systems are closely linked to this formalism. Even though XMG is dedicated to the production of different kinds of grammar, some of its features only make sense when dealing with TAG. Among these features XMG integrates a library of *linguistic principles* that XMG's user may activate to ensure that the produced trees respect the TAG formalism. Among these principles there are the clitic ordering, the uniqueness of extraction and clitic rank, and the possibility to assign color to nodes to automatize the fragment combination (see Controlling fragment combination semi automatically by coloring nodes).

3.1.5 To sum up

XMG formalism is a declarative logical language of conjunctions and disjunctions that allows one to express directly how abstractions are to be combined. These abstractions are also called **classes** and contain tree fragments when dealing with TAGs. These classes are arranged in a multiple inheritance hierarchy, so that one can flexibly reuse and specialize information. As a result one can finely describe the generalizations that appear in a grammar. The high degree of factorization given by XMG eases grammar development and maintnance (update done by modifying specific classes).

3.2 A closer look at a metagrammar

Here, we will see exactly what kind of information a metagrammar contains, at the same time introducing XMG's concrete syntax. After that we will be able to start writing our own metagrammar.

3.2.1 Principles

The first piece of information one has to give in a metagrammar is the principles that will be needed to compute the grammar structures. The instruction used to do this is the **use *principle* with (*constraints*) dims (*dimensions*)** statement. For instance, one may decide to force the syntactic structures of the output grammar to have the grammatical function *gf* with the value *subj* only once. This is told by:

```
use unicity with (gf = subj) dims (syn)
```

In the **syn** dimension, we use the **unicity** principle on the attribute-value **gf = subj**. At the time of this writing 3 principles are available in the XMG system, namely:

unicity	uniqueness on a specific attribute-value
rank	ordering of clitics by means of associating the *rank property* to nodes
color	automatization of the node merging by assigning color to nodes

Note that principles use as parameters pieces of information that are associated to nodes with the status *property* (see below).

3.2.2 Types, properties and features

XMG includes light typing. By "light" we mean that one has to type the pieces of information that are used, but for now there is no strong type checking during compilation and execution (but only a syntax checking). There are 4 ways of defining types:

- as an enumerated type, using the syntax **type *Id* = {*Val1*,...,*ValN*}** such as in:

```
type CAT={n,v,p}
```

(note that the values associated to a type are *constants*)

- as an integer interval, using the syntax **type *Id* = [*I1* .. *I2*]** such as in:

```
type PERS=[1 .. 3]
```

- as a structured definition (T1 ... Tn represent types) **type *Id* = [*id1* : *T1* , *id2* : *T2* , ..., *idn* : *Tn*]**, such as in:

```
type ATOMIC=[
     mode : MODE,
     num : NUMBER,
     gen : GENDER,
     pers : PERS]
```

- as an unspecified type **type *Id* !**, such as in:

```
type LABEL !
```

(this is useful when one wants to avoid having to define acceptable values for every single piece of information). Note that XMG integrates 3 predefined types: **int**, **bool** (whose associated values are + and -) and **string**.

Once types have been defined, we can define *typed properties* that will be associated to the nodes used in the tree descriptions. The role of these properties is either (a) to provide specific information to the compiler so that additional treatments can be done on the output structures to ensure their well-formedness or (b) to decorate nodes with a label that is linked to the target formalism and that will appear in the output (see XMG's graphical output). The syntax used to define properties is **property *Id* : *Type***, such as in:

```
property extraction : bool
```

There also exists a syntactic sugar concerning properties. Here one may want to avoid having to state *extraction*=+ several times. An alternative to this is to associate an *abbreviation* (between curly-brackets):

```
property extraction : bool {extra = +}
```

This means that using *extra* is equivalent to giving the value + to the property *extraction* of a node, ie equivalent to *extraction*=+.

Eventually we have to define *typed features* that are associated to nodes in several syntactic formalisms such as Feature-Based Tree Adjoining Grammars (FBTAG) or Interaction Grammars (IG). The definition of a feature is done by writing **feature *Id* : *Type***, such as in:

```
feature num : NUMBER
```

Up to now, we have seen the declarations that are needed by the compiler to perform different tasks (syntax cheking, output processing, etc). Next we will see the heart of the metagrammar: the definition of the `clauses`, ie the **classes**.

3.2.3 Classes

Here we will see how to define classes (i.e. the *abstractions* in the XMG formalism). Note that in TAG these classes refer to tree fragments. A class always begins with **class *Id***, such as in:

```
class CanonicalSubj
```

N.B. A class may be *parametrized*, in that case the parameters are bracketted and separated by a colon, as presented in Miscellaneous.

3.2.3.1 Import

To reach a better factorization, a class can *inherit* from another one. This is done by invoking **import *Id*** (where Id is a class name), such as in:

```
import TopClass[]
```

That is to say, the metagrammar corresponds to an **inheritance hierarchy**. But what does inherit mean here ? In fact, the content of the imported class is made available to the daughter class. More precisely, a class uses *identifiers* to refer to specific pieces of information. When a class inherits from another one, it can reuse the identifiers of its mother class (provided they have been *exported*, see below). Thus, some node can be specialized by adding new features and so on.

Note that XMG allows *multiple inheritance*, and besides it offers an extended control of the scope of the inherited identifiers, since one can restrict the import to specific identifiers, and also rename imported identifiers (see Miscellaneous).

N.B. When importing a class, even if it has parameters in its definition, these *cannot be instantiated*.

3.2.3.2 Export

As we just saw, we use in each class **identifiers**. One important point when defining a class is the scope we want these identifiers to have. More precisely we can give (or not) an extern visibility to each identifier by using the `export` declaration. Only exported identifiers will be available when inheriting or calling (ie instantiating) a class. Identifiers are exported using **export *id1 id2 ... idn*** such as in:

```
export ?X ?Y
```

(The *?* indicated X and Y are variables, and not skolem constants, ie anonymous constants that would have been prefixed with *!*) Besides, when exporting an identifier you can rename it so that it can later be referred to by a new name (to avoid name conflict). This is done by typing **export *id1=id1new***, example:

```
export ?X=?U ?Y
```

(here the X variable will be referred to by using *?U* in the daughter class, ?Y will still be called ?Y)

3.2.3.3 Identifiers

In XMG, identifiers can refer either to a node, the value of a node property, or the value of a node feature. But whatever an identifier refers to, it must have been declared before by typing **declare *id1 id2 ... idn***, such as in:

```
declare ?X ?Y ?Z
```

Note that in the *declare section* the prefix *?* (for variables) and *!* (for skolem constants) are **mandatory**.

3.2.3.4 Content

Once the identifiers have been declared and their scope defined, we can start describing the content of the class. Basically this content is given between curly-brackets. This content can either be:

- a *statement*,
- a **conjunction** of *statements* represented by "**`S1 ; S2`**" in the XMG formalism,
- a **disjunction** of *statements* represented by "**`S1 | S2`**",
- or a *statement* associated to an **interface** (see below).

By *statement* we mean:

- an *expression*: **`E`** (that is a **variable**, a **constant**, an **attribute-value matrix**, a **reference** (by using a **dot** operator, see the example below), a **disjunction of expressions**, or an **atomic disjunction of constant values** such as **`@{n,v,s}`**),
- a **unification equation**: **`E1=E2`**,
- a **class instanciation**: **`ClassId[]`** (note that the square-brackets after the class id are mandatory even if the instantiated class has no parameter),
- or a **syntactic description**.

A syntactic description is given following the pattern **<syn>{ *formulas* }**. Now what kind of formulas doas a syntactic description contain ? The answer is *nodes*. These nodes are in relation with each other. In XMG, you may give a name to a node by using a variable, and also associate properties and/or features with it. The classic node definition is **node *?id* (*prop1=val1 , ... , propN=valN*) [*feat1=val1 , ... , featN=valN*]** such as in:

```
node ?Y (gf=subj)[cat=n]
```

Here we have a node that we refer to by using the ?Y variable. This node has the property *gf* (grammatical function) associated with the value *subj*, and the feature structure *[cat=n]* (note that associating a variable to a node is *optional*).

Once you defined the nodes of the tree fragment, you can describe how they are related to each other. To do this, you have the following operators:

->	strict dominance
->+	strict large dominance (transitive non-reflexive closure)
->*	large dominance (transitive reflexive closure)
>>	strict precedence
>>+	strict large precedence (transitive non-reflexive closure)
>>*	large precedence (transitive reflexive closure)
=	node equation

Each subformula you define can be added conjunctively (using "**;**") or disjunctively (using "|") to the description. For instance, the fragment introduced previously, that is:
Tree fragment

can be represented by the following code in XMG:

```
class Example
declare ?X ?Y ?Z
{<syn>{
      node ?X [cat=S] ; node ?Y [cat=N] ; node ?Z [cat=V] ;
      ?X -> ?Y ; ?X -> ?Z ; ?Y >> ?Z
      }
}
```

XMG also supports an alternative way of specifiyng how the nodes are related to each other. This alternative syntax should allow the user to both define the nodes and give their relations at the same time:

node { node }	strict dominance
node { ...+node }	strict large dominance (transitive non-reflexive closure)
node { ...node }	large dominance (transitive reflexive closure)
node node	strict precedence
node ,,,+node	strict large precedence (transitive non-reflexive closure)
node ,,,node	large precedence (transitive reflexive closure)
=	node equation

Thus the tree fragment above could be defined in the XMG syntax the following way:

```
class Example
{<syn>{
       node [cat=S] {
              node [cat=N]
              node [cat=V]
              }
       }
}
```

Note that the use of variables to refer to the nodes becomes useless inside the fragment, nonetheless we may want to assign variables to node to reuse them later through inheritence.

3.2.3.5 Interface

interfaces can be seen as a third type of class content (the 2 others being **syn** and **sem**). They corresponds to an attribute-value matrix, allowing one to associate a *global name* to an identifier.

The syntax of the interface is the following (the interface is between square-brackets): **class *Id* { ... } *= [*Name1=Id1, ... , NameN=IdN*]**. The ***=** operator represents *unifying extension*.

When a class is valuated, the descriptions (contained in the classes) it refers to are accumulated. At the same time, the interfaces associated with these descriptions are accumulated. The semantics of their accumulation may correspond to unification.

More precisely, there are several types of interface accumulation that are available in XMG (see the table below). Nonetheless, the unifying extension reveals itself to be the most useful one. It performs an unification of the interfaces (ie open feature structures) when 2 fragments are either combined (using class intanciations) or in an inheritence relation.

::	interface constraint
:=	interface assignment
+=	interface disjoint extension
-=	interface attrition

Let us see the use of an interface in an example. Considering the tree fragment used so far. Imagine we want to refer to the N node outside of the class. To do so, we give this node a global name. We can do this by using the following interface:

```
class Example
declare ?X ?Y ?Z
{<syn>{
       node ?X [cat=S] {
                         node ?Y [cat=N]
                         node ?Z [cat=V]
                        }
       }*=[subj = ?Y]
}
```

In a class `A` which is combined with `Example`, you can constraint the identification of a local node X with the subj node of `Example` by reusing the feature *subj* in the interface of `A` :

```
*=[subj=?X]
```

Note that the interface may also be used to give names to properties or feature values.

3.2.4 Valuations

Once all the classes have been defined, we can ask for the evaluation of the classes that will trigger the combination of the fragments (ie classes calling classes that contain disjunction and/or conjunction of fragments). For each of these specific classes, we will obtain an accumulated tree description that may lead to the building of 0, 1 or more TAG trees. The syntax of the evaluation instruction in XMG is **value *Id***, such as in:

```
value n0Vn1
```

3.3 XMG's concrete syntax: TAG example

Now, we will see in details how to write a metagrammar. We will define a metagrammar generating a small TAG for French. This small TAG will contain 2 trees, namely the ones representing a transitive verb either with a canonical subject or a subject in relative position.

3.3.1 Specifying data

First thing to do: defining the principles, types, properties and features we will use. For sake of clarity, we will only constraint the produced trees to have no duplicate grammatical function. That is to say, we will only activate the *unicity principle* with the *gf* property as parameter:

```
use unicity with (gf = subj) dims (syn)
use unicity with (gf = obj) dims (syn)
```

We will deal with few types in this example. We only pay attention to *grammatical functions* and *syntactic categories*. The first one is a node property and the second one a node feature (ie part of the TAG formalism):

```
type CAT = {n,v,s}
type GF = {subj, obj}

property gf : GF
feature cat : CAT
```

3.3.2 Defining blocs (tree fragments)

The metagrammatical rule we will use is the following:

$$\text{transitive} = (\text{CanSubject} \vee \text{RelSubject}) \wedge \text{Active} \wedge \text{Object}$$

So we will handle 4 tree fragments: `Active`, `CanSubject`, `Object`, and `RelSubject`. The class `transitive` will consist of an abstraction on a conjunctive combination including a disjunction on the subject that is used.

The `Active` class corresponds to the verbal spine:

Tree fragment

```
class Active
export ?X ?Y
declare ?X ?Y
{<syn>{
        ?X -> ?Y
        }
}
```

The `CanSubject` class corresponds to the Example class introduced previously:

Tree fragment

```
class CanSubject
export ?X ?Y ?Z
declare ?X ?Y ?Z
{ <syn>{
        node ?X [cat = s]{
                node ?Y (gf=subj)[cat=n]
                node ?Z [cat = v]
                }
        }
}
```

The `Object` class is the symetric class of `CanSubject`:

Tree fragment

```
class Object
export ?X ?Y ?Z
declare ?X ?Y ?Z
{ <syn>{
        node ?X [cat = s]{
                node ?Y [cat = v]
                node ?Z (gf=obj)[cat=n]
                }
        }
}
```

The `RelSubject` class and its concrete syntax are given below:

Tree fragment

```
class RelSubject
export ?X ?Y ?Z
declare ?X ?Y ?Z ?U ?V
{ <syn>{
        node ?U [cat = n]{
                node ?V [cat = n]
                node ?X [cat = s]{
                        node ?Y (gf=subj)[cat=n]
                        node ?Z [cat = v]
                        }
                }
        }
}
```

At this point, we may wonder why associating variables to nodes ? The answer is that we still have to merge these fragments, we will use the exported variables to unify specific nodes.

3.3.3 From tree fragments to trees

Once the basic blocs have been defined, we can combine them to produce the expected trees. We define the `transitive` class:

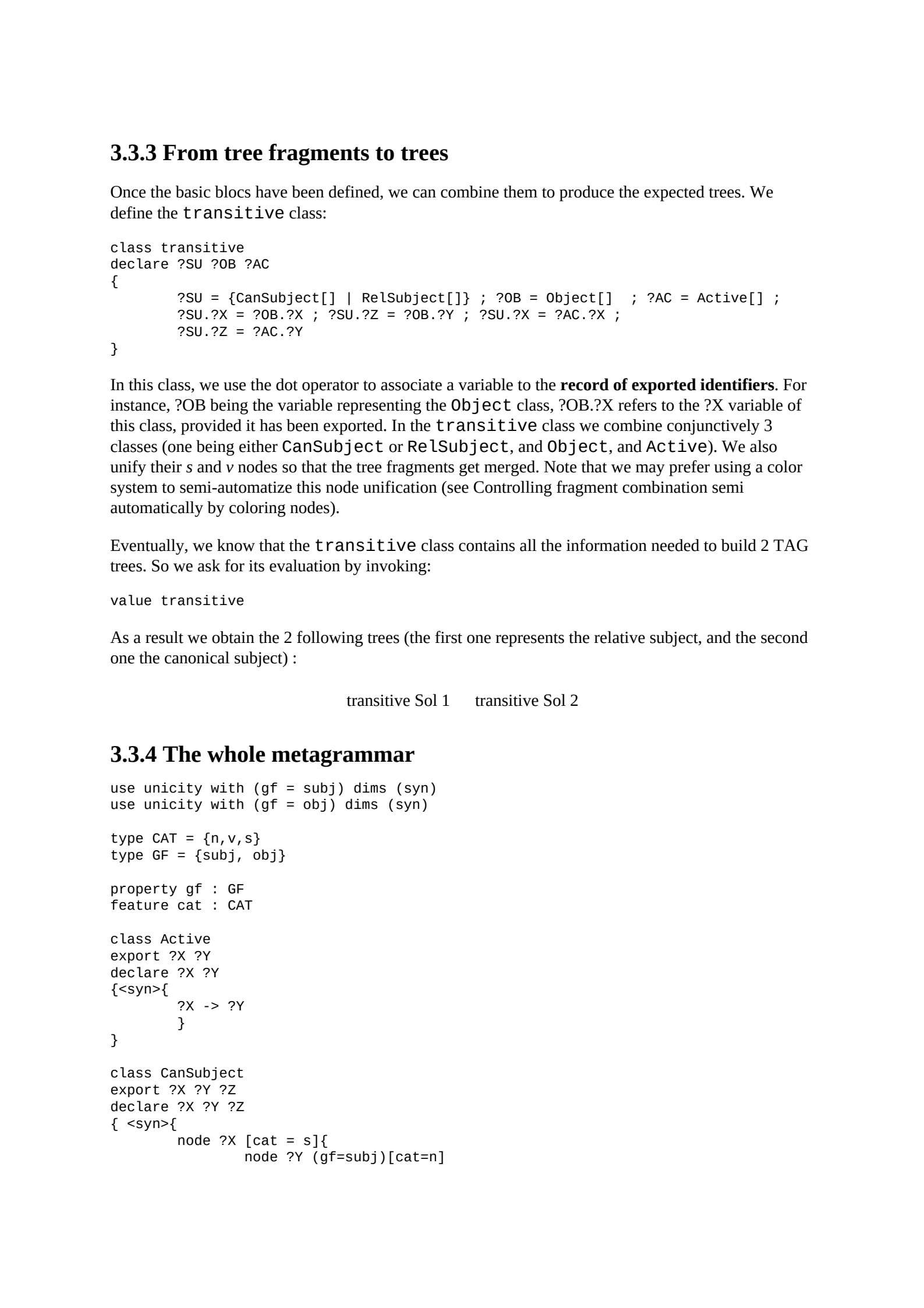
```
class transitive
declare ?SU ?OB ?AC
{
        ?SU = {CanSubject[] | RelSubject[]} ; ?OB = Object[]  ; ?AC = Active[] ;
        ?SU.?X = ?OB.?X ; ?SU.?Z = ?OB.?Y ; ?SU.?X = ?AC.?X ;
        ?SU.?Z = ?AC.?Y
}
```

In this class, we use the dot operator to associate a variable to the **record of exported identifiers**. For instance, ?OB being the variable representing the `Object` class, ?OB.?X refers to the ?X variable of this class, provided it has been exported. In the `transitive` class we combine conjunctively 3 classes (one being either `CanSubject` or `RelSubject`, and `Object`, and `Active`). We also unify their *s* and *v* nodes so that the tree fragments get merged. Note that we may prefer using a color system to semi-automatize this node unification (see Controlling fragment combination semi automatically by coloring nodes).

Eventually, we know that the `transitive` class contains all the information needed to build 2 TAG trees. So we ask for its evaluation by invoking:

```
value transitive
```

As a result we obtain the 2 following trees (the first one represents the relative subject, and the second one the canonical subject) :



3.3.4 The whole metagrammar

```
use unicity with (gf = subj) dims (syn)
use unicity with (gf = obj) dims (syn)

type CAT = {n,v,s}
type GF = {subj, obj}

property gf : GF
feature cat : CAT

class Active
export ?X ?Y
declare ?X ?Y
{<syn>{
        ?X -> ?Y
        }
}

class CanSubject
export ?X ?Y ?Z
declare ?X ?Y ?Z
{ <syn>{
        node ?X [cat = s]{
                node ?Y (gf=subj)[cat=n]
```

```
                node ?Z [cat = v]
                }
        }
}

class Object
export ?X ?Y ?Z
declare ?X ?Y ?Z
{ <syn>{
        node ?X [cat = s]{
                node ?Y [cat = v]
                node ?Z (gf=obj)[cat=n]
                }
        }
}

class RelSubject
export ?X ?Y ?Z
declare ?X ?Y ?Z ?U ?V
{ <syn>{
        node ?U [cat = n]{
                node ?V [cat = n]
                node ?X [cat = s]{
                        node ?Y (gf=subj)[cat=n]
                        node ?Z [cat = v]
                        }
                }
        }
}

class transitive
declare ?SU ?OB ?AC
{
        { ?SU=CanSubject[] | ?SU=RelSubject[] } ; ?OB = Object[]  ; ?AC = Active[] ;
        ?SU.?X = ?OB.?X ; ?SU.?Z = ?OB.?Y ; ?SU.?X = ?AC.?X ;
        ?SU.?Z = ?AC.?Y
}

value transitive
```

3.4 Extension to other descriptive levels (semantics, etc): handling dimensions

Up to now, we have seen how to design a rather simple metagrammar. Now imagine we would like to extend it so that we can describe other kinds of information that only syntactic trees. To distinguish each level of description we manage, we will use *dimensions*. A dimension is characterized by a specific sublanguage, and a combination operation. For instance, when dealing with tree fragments, the semantic of the combining process is (a) to *accumulate* the description and (b) perform unification between data structures (nodes, features, etc). At the time of this writing, 3 dimensions are available in the XMG system:

<syn>	*Syntactic dimension*: tree descriptions (for describing TAG tree fragments or IG tree descriptions)
<sem>	*Semantic dimension*: predicative structures (*Flat semantics*, see below)
<dyn>	*Dynamic dimension*: previously called *interface* in this documentation (formally equivalent to a dimension)

3.4.1 Predicate semantics

Here we will see how to describe semantic information. Basically, this dimension allows one to describe:

- predicates with 0, 1 or more arguments and a label,
- negation,
- a specific relation called "scope_over" for dealing with quantifiers,
- and semantic identifiers.

So the language of the semantic dimension is:

$$\text{Description} ::= \ell\{:\}p(E_1,...,E_n) \mid \neg \ell\{:\}p(E_1,...,E_n) \mid E_i << E_j \mid E$$

In XMG concrete syntax, one may define a class with a semantic content by:

```
class BinaryRel
declare !L ?X ?Y ?P
{ <sem>{!L:?P(?X,?Y)
       }
}*=[pred=?P]
```

That is to say, we define the class `BinaryRel` in which 3 variables and a skolem constant (prefixed by "!") are declared. This class only contains semantic information (dimension `<sem>`), more precisely it contains a predicate (whose value is the variable ?P) of arity 2, its arguments are the variables ?X and ?Y. !L represents the label associated to this predicate. Note that we use the interface dimension to give the name *pred* to ?P. Further, this variable may be unified with a constant, and the value of the predicate thus given.

Finally, it is possible to define a class containing both a semantic and syntactic dimension, and these dimensions may share identifiers. Besides sharing identifiers may also be done by using the interface dimension. Thus XMG provides efficient devices to define a syntax / semantics interface within the metagrammar.

3.4.2 Tree schematas / Lexicon interface

Wide-coverage TAGs may contain thousands of trees, which often share their structure (only the lexical item and its morpho-syntactical information differ). So a metagrammar compiler does not produce whole trees, but *tree schematas*, that is to say trees with a distinguished node called the *anchor* and which refers to the place where the lexical item will be put before parsing. Recent works have pled for using an attribute-value matrix (AVM) associated to each tree, and which would contain the information needed to perform lexical selection. This AVM is called *HyperTAG*. Through the *interface* dimension, users can define such hyperTAGs with XMG.

3.5 Using XMG to produce IG

Some information about Interaction Grammars is available at
http://www.loria.fr/equipes/calligramme/leopar/doc.html.

Writing IGs with XMG is not so different from writing TAGs so all previous information holds unless stated otherwise. Instead of running MetaTag, you have to run MetaIG. Simple, isn't it !

There isn't any built-in graphical front-end for IGs, you must use Leopar to view your Grammar ! A typical command line scenario is

MetaIG grammar.mg -o grammar.xml

That line generates a xml file that can be viewed with leopar.

3.5.1 No principles

If you plan to write IGs, you can't use principles such as colors, unicity ... These principles are triggered at the solving step when XMG constrains Tree Descriptions to Trees and this step is not performed by MetaIG. They are 2 main consequences:

On the one hand, writing IGs is simpler than writing TAGs, because you don't have to bother with principles like for example checking that all the nodes are colored. The solutions are Tree Descriptions and are very close to what the grammarian writes in XMG, so debugging is easier.

On the other hand, no unifications are performed implicitly, so you have to pay attention to node unification that you would have delegated to colors with TAGs.

3.5.2 Polarised Features

In IGs node features are polarised. This means that features are not couples (feature_name,feature_value) but triplets (feature_name,polarity,feature_value) where polarity is either positive, negative, neutral or virtual. So there are 4 symbols to write this:

positive: feat =+ val
negative: feat =- val
neutral: feat = val
virtual: feat =~ val

3.6 XMG's outputs

3.6.1 Output content

What does XMG exactly produce ? The answer is a list of *entries*. Each of these contain:

- a **unique name** of the form *`TFamilyName-Id`*,
- a **family name** (the name of the class which triggered the compilation, ie the valuated class which corresponds to the query of our formalism),
- a **trace**, which corresponds to the list of the classes that have been accumulated to produce this entry,

- a **syntactic description** (a tree schemata for TAG, a tree description for IG),
- a **semantic description** (possibly empty),
- an AVM corresponding to the **interface** (possibly empty),
- and a **trace summary** (only in the GUI), see Using trace summaries.

Furthermore, through the GUI you can allow features display, and also define the list of features you want to print on screen.

3.6.2 Output formats

There are different output formats in XMG. The automatically produced grammars can either be printed on screen through a Qtk graphical interface (for TAG), printed in an XML file, or saved in an internal Oz format for further processing (see Pickle). The advantage of the first output is that it provides an important help during grammars development, whereas the second one (XML) offers the ability to be converted relatively easily to be reused within other NLP applications such as parsers or generators, and the third one provides a representation that can be efficiently processed by another Oz program (see Using the XMG-TOOLS).

Note that the GUI is only available for TAG, since the viewer for IG is included in the LEOPAR parser developed at LORIA by the Calligramme Project (see http://www.loria.fr/equipes/calligramme/leopar/).

NB:

- the DTD for the XML formats (TAG and IG) are included in the XMG system and available at the following address:

https://sourcesup.cru.fr/docman/index.php?group_id=99

- the XMG commands and options are given in List of XMG's options.

4 XMG's architecture

In this section we will explain how the XMG system has been implemented, introducing its 3-part architecture. These are:

1. the *compiler front-end* that compiles the object-oriented concrete syntax into XMG's core language,
2. the *virtual machine (VM)* that executes the instructions corresponding to the core language,
3. the *third-part* that performs additional treatments on the VM's output (TAG validity checkings, etc).

4.1 The compiler front-end

The compiler's role is to translate metagrammatical descriptions into instructions of the core language. It is implemented using "Gump - the Front-End Generator for Oz". Gump is a tool that takes as input a lexer/parser specification and produces the corresponding Oz code (it is based on the GNU tools *flex* and *bison*).

XMG's compiler performs the following tasks:

- syntax checking of the metagrammatical description (static checking),
- resolution of the identifiers' scope and identifiers' renaming (processing of the `import` and `export` declarations),
- compilation of the concrete code into instructions for the virtual machine (including description formulas' unfolding).

The following files are part of the compiler module:

```
Lexer.oz
Parser.oz
SyntaxChecker.oz
Normalizer.oz
ClassMG.oz
VMInstructions.oz
Compiler.oz
DescClass.oz
Unfolder.oz
Valuer.oz
```

4.2 The virtual machine

The virtual machine (VM) implements a standard logic programming kernel with chronological backtracking. It is inspired by the Warren Abstract Machine, but unlike this, it uses *structure sharing*, ie each term is represented by a pair of a pattern and an environment in which to interpret it. The advantage of the *structure sharing* technique is that it enables to save memory space.

Besides, the VM is implemented using object-oriented programming. The prototype of each of its method corresponds to an instruction, so it can directly execute the code produced by the compiler module. The use of object-oriented programming makes the VM easier to extend. The extension of the VM can be due either to the definition of a new dimension that will be *accumulated* separately, or to the use of non-standard data types which need specific unification (such as polarised features in IG).

The VM computes all possible derivations for a valuation statement, then takes a snapshot of its accumulators. This snapshot contains a tree description for the syntactic dimension, and optionally a list of semantic formulas. When working with TAG, the tree description needs to be solved in order to find out all the trees that respect it. When working with IG, no further processing is needed.

The VM corresponds to the files :

```
Engine.oz
Polarity.oz
```

4.3 The third part

As we saw previously, the VM produces snapshots of its accumulators. It may happen that these snapshots need to be further processed. It is the case for TAG, as we do not want tree descriptions but trees, more precisely we want all the models that satisfy the syntactic description. So the first processing made on the VM's output is a **description solving**. Because of the high complexity of this satisfiability problem, we chose a *constraint-based approach* to reduce the search space. The idea of XMG's description solver is to convert each relation between nodes in term of constraints on sets of integers. First we associate an integer to each node of the description. Then we pose constraints in

accordance with the relations that hold between 2 nodes. For instance, when a node *i* dominates a node *j*, we can pose the constraint that:

- the set of all the nodes that are equal to or above *i* in the model contains all the nodes that are strictly above *j*

and

- the set of all the nodes that are strictly below *i* in the model is included in the set of the nodes that are below *j*

and

- the set of all the nodes that are strictly on the left of *i* in the model contains the nodes that are on the left of *j*

and

- all the nodes that are strictly on the right of *i* in the model are included in the set of the nodes that are on the right of *j*.

This conversion corresponds to the following formula:

Constraints on sets of integers

Besides we use a *modular* description solver. By modular, we mean that the solver is composed of different modules solving different kinds of constraints:

- tree well-formedness constraints (such as those given above),
- principle-based constraints (constraints linked to node coloring, the unicity principle, or the clitic ordering).

The files that are part of the solver module are:

```
Prepare.oz
SolverObj.oz
SolverDesc.oz
ColorPrinciple.oz
RankPrinciple.oz
ExtractionPrinciple.oz
```

Once we have produced (a) trees for TAG or (b) tree descriptions for IG, we can send these to the output formatting process. For TAG, 2 output formats are available: a Graphical User Interface or an XML writer. For IG, only the second format is available. The files included in the formatting are:

```
Decoder.oz
XMLconvert.oz
Printer.oz
TracePrinter.oz
DisplayTrees.oz
OutputGUI.oz
```

Finally, the *main* files of the XMG system (technically speaking) are:

```
MetaTAG.oz
MetaIG.oz
```

These files contain the code that instanciates the different modules and thus performs the processings on the metagrammar. They correspond to the 2 executables provided by XMG, namely **`MetaTAG`** and **`MetaIG`**, to compile respectively TAG and IG metagrammars.

5 Advanced Topics

In this section, we will discuss different topics that you do not need to know about to use the XMG system, but that may give a help to develop metagrammars.

5.1 Controlling fragment combination semi automatically by coloring nodes

The first topic concerns the use of a color language to semi-automatize node unification during tree description solving. This idea has been proposed by B. Crabbé (see [Crabbé and Duchier, 04b]). The process is the following:

1. we decorate nodes with colors (red, black or white),
2. the description solving is extended so that the nodes are unified according to specific color combination rules:

Color mergnig table

That is to say:

- a black node may be unified with 0, 1 or more white nodes and thus produces a black node,
- a white node has to be unified with a black one producing a black node,
- and eventually a red node cannot be merged with any other node.

As a result, a satisfying model is a model where all the nodes are either *black* or *red*.

The important advantage of this color labelling process is that we do not need to expicitly specify all the node unifications that have to be performed. Actually the saturation of colors will trigger these unifications. In other words we can think of nodes in terms of "relative addresses". This means that we do not have to manage node variables (which correspond to "absolute addresses") as the colors give a way to refer to "mergeable" nodes, ie black nodes that can be unified (thus that can receive a fragment).

By lessening the use of variables, we prevent name conflicts and thus we can for instance easily reuse the same tree fragment within the same tree description (this happens in TAG for trees with double prepositional phrase).

5.2 Using the XMG-TOOLS

The XMG system comes not alone as several additional tools are available on the xmg project page. There are 5 XMG-TOOLS at the moment of this writing, namely:

- the **classbrowser**: a perl/Gtk2 module offering a view of the metagrammar classes with the possibility to comment them (similar to the javadoc utility),
- the **hierarchy_printer**: a perl module allowing to extract a graphical view from the metagrammar (hierarchy of classes and variables handled). Courtesy of Ulrike Fleury.
- the **CHECKER**: a module implemented in Oz and whose role is to check if there are duplicated trees among a grammar (that is, identical trees that have been build using different classes),
- the **VIEWER**: also implemented in Oz (using the Qtk library), this module searches for specific trees among the grammar from a tree name or the class(es) that belongs to their trace,
- the **SELECTOR**: from an anchoring description (text file respecting a specific syntax, see http://wiki.loria.fr/wiki/XMG/SELECTOR) and a grammar, produces lexicalised TAG trees (ie trees with lexical item(s) as leaves).

5.2.1 Using the classbrowser

This tool is a Metagrammar class Browser. It reads a Metagrammar source file and displays its content in a Browser similar to Gnome Help.

- Requirements :

```
 - Perl version >= 5.8.0
 - Gtk2
 - Gtk2/Perl bindings
```

- Usage :

```
> perl main.pl [mgfile]
```

or :

```
> chmod 755 main.pl
> ./main.pl [mgfile]
```

- Comments in the Metagrammar file :

In order to get an enhanced view of the metagrammar, modify the metagrammar source comments the following:

  - Introduce comments that you wish to get in the comments section of the browser with %#
  - Introduce Metagrammar sections with the line of comments %*

- Examples :

```
% Usual Metagrammmar comment
%# The following comment will appear in the class which is currently defined
%* The following comment introduces a Metagrammar section.
```

5.2.2 Using the hierarchy printer

This perl programme extract a pdf view of the graph of the class hierarchy.

- Requirements :

```
  - Perl version >= 5.8.0
  - Graphviz
```

- Usage :

```
> perl hierarchie_classes.pl [mgfile]
```

or :

```
> chmod 755 hierarchie_classes.pl
> ./hierarchie_classes.pl [mgfile]
```

5.2.3 How to remove the duplicated trees: using the CHECKER

The CHECKER module provides the **`CheckTAG`** program which can be used to remove duplicated trees (ie identical trees that are produced by combining different tree fragments). To install it, just download the sources from Subversion repository (see http://sourcesup.cru.fr/xmg), go in the local sources directory, and invoke:

```
ozmake --upgrade
```

To use it, you need to first compile your metagrammar producing a Pickle as an output (see List of XMG's options for the command to use). Then you can use the CHECKER to perform different tasks according to the options used:

1. no option: statistical use, it returns the number of duplicated trees in the grammar,
2. --gui: it prints the duplicated trees sorted by family name in the same Qtk interface as XMG,
3. --xml: it prints the grammar without the duplicated trees in an XML format (respecting XMG's DTD for TAG), used with -m <Int> allows to split the grammar in several XML files,
4. --rec: it records the grammar as a Pickle (Oz internal representation).

- Usages:

```
CheckTAG GrammarFile.rec
CheckTAG GrammarFile.rec --gui
CheckTAG GrammarFile.rec --xml -o FilteredGrammarFile.xml
CheckTAG GrammarFile.rec --xml -m <Int> -o XMLCheckedGrammarRoot
CheckTAG GrammarFile.rec --chk -c FilteredGrammarFile.rec
```

5.2.4 How to search for specific trees: using the VIEWER

The VIEWER module offers a graphical interface allowing one to search for specific trees in the produced grammar, either from a tree name or a list of fragments (ie class names). It can be installed by downloading the sources, going in the sources directory, and invoking:

```
ozmake --upgrade
```

It provides the **`ViewTAG`** program, which can be used in several ways:

- with no argument: you will load the grammar in a Pickle format (see List of XMG's option for the command to use to produce grammars in this format) through a browser and enter the search criteria in the interface,
- with a grammar file as first argument: you will enter the search criteria in the interface,

- with a grammar file as first argument and a tree name as second argument: a window will appear with the searched tree if it exists or an error message if it does not.

- Usages:

```
ViewTAG
ViewTAG GrammarFile.rec
ViewTAG GrammarFile.rec TreeName
```

5.2.5 How to anchor TAG tree schematas: using the SELECTOR

The SELECTOR module is an anchoring module, taking as input a grammar in a binary format (Pickle) and a specification of lexical entries following a predefined syntax, and producing as output anchored TAG trees. The installation procedure is the following: first download the sources, go to the sources directory, and then invoke:

```
ozmake --upgrade
```

It provides the **`SelectTAG`** program, which performs anchoring from a grammar (Pickle) and a filter (ie a specification of the lexical entries to use, along with their morpho-syntactical informations). There are several usages of this program:

- with the --gui option: it prints on the screen the *anchored trees* (in the same Qtk graphical interface as XMG's),
- with the --xml option: it prints the anchored trees in an XML file (respecting XMG's DTD for TAG),
- with the --chk option: it prints the anchored trees in an Oz internal representation format (Pickle).

- Usages:

```
SelectTAG GrammarFile.rec FilterFile --gui
SelectTAG GrammarFile.rec FilterFile --xml -o OutputFile
SelectTAG GrammarFile.rec FilterFile --chk -c OutputFile
```

5.3 Interfacing XMG with Natural Language Parsers

The XMG system is used to develop linguistic resources. These resources are produced in an XML format so that they can be easily used in NLP applications. We will see here how to use the semi-automatically produced grammars in parsing, in particular we will see how to use the TAG grammars with the DyALog and LLP2 systems, and the IG grammars with the LEOPAR system.

5.3.1 Dealing with TAG

Several TAG parsers are available, we have worked on interfacing the XMG system with 2 of these, namely the DyALog system and the LLP2 system (which are both freely available).

5.3.1.1 DyALog

The DyALog system is developed at the INRIA Rocquencourt Laboratory, within the ATOLL project, by Eric De La Clergerie. It consists of a compiler for logic programs using tabulation techniques. Since release 1.1, it can be used to compile tabular parsers for TAG. It uses a XTAG-like architecture where the resources are splitted in:

- a tree schematas lexicon,
- a list of lemmas,
- a morphological lexicon.

With XMG, one can produce the needed tree schematas in XML, and then convert them to DyALog's XML format by using the XSLT program available at http://sourcesup.cru.fr/xmg/loria2tagml.xsl.

5.3.1.2 LLP2

The Loria LTAG Parser 2 is developed by Azim Roussanaly at the LORIA Laboratory, within the Langue Et Dialogue project. It consists of an evolution of Patrice Lopez' TAG parser. It is implemented in Java, and provides a TAG parser usable either in command line or with a graphical interface. The resources it needs adopt an XTAG-like architecture composed of:

- a tree schematas lexicon,
- a list of lemmas,
- a morphological lexicon.

One can produce the tree schematas with XMG, and then convert them to LLP2's XML format by using the XSLT program available at http://sourcesup.cru.fr/xmg/mg2tagml.xsl.

5.3.2 Dealing with IG

5.3.2.1 LEOPAR

For more information, please consult LEOPAR's web site.

5.4 Interfacing XMG with Natural Language Generator(s)

5.4.1 GenI

GenI is a surface realiser. It produces natural language sentences from a TAG grammar and a flat semantic input. More precisely, it requires:

- a set of tree schematas
- a lexicon
- an input semantics or test suite

Note: These requirements are not yet stable. For the most up to date information, see the GenI documentation.

Tools you need:

- MetaTAG
- Selector
- ViewTAG

Things which are different from the basic GenI usage:

- you write an XMG metagrammar instead of .geni trees
- you write a .lex lexcion instead of the normal GenI format

Procedure: Run MetaTAG to produce a pickle of the grammar. See the common grammar manifesto for details on how to deal with the lexicon. Assuming you set up GenI and its index files correctly, GenI will take care of the rest.

6 Miscellaneous

In this section we will discuss different aspects of the XMG system, going from its availability to the list of its options.

6.1 Availability of the XMG system (License, etc)

The XMG system is freely available at http://sourcesup.cru.fr/xmg under the terms of the CeCILL license (GPL compliant).

6.2 Including files in the metagrammar

To ease metagrammar development, a *file inclusion* process is integrated in XMG. You can split a metagrammar in several files by using the instruction **include *file.mg*** such as in:

```
include header.mg
class CanSubj ...
```

Note that the file path is either relative to the file which is given as an argument to `MetaTAG / MetaIG` or absolute. Note also that it is only a substitution instruction, so it can appear anywhere in the main metagrammar file, it will be substituted before compilation. Keep in mind that the order of the information in the metagrammar is sensitive, for instance the type declarations have to appear before the class definitions.

N.B.: note that this include instruction also allows you to share common parts between different metagrammars.

6.3 Splitting the produced grammar in several files

You can also split the produced grammar in several XML files. To do so, use the *-m* options (m stands for *many*) followed by an integer representing the number of files.

6.4 Using preprocessing macros

In XMG you can use preprocessing macros to subsitute specific strings in the metagrammar file. These macros are used by invoking **macro *String NewString***, such as in:

```
macro genv @{aux,v}
```

Here we define a macro that will replace all occurences of the **genv** string by **@{aux,v}** (which represents the atomic disjunction of the constant values aux and v).

Further in the metagrammar, one can thus define a node by:

```
node [cat = genv]
```

which is equivalent to:

```
node [cat = @{aux,v}]
```

N.B.: as macros are processed before compilation and correspond to string substitutions, they can be defined everywhere in the metagrammar file, provided they are defined before they are used in the description.

6.5 Using trace summaries

To ease metagrammar development, it is possible to put a highlight on specific classes that will appear in the graphical output, in the **trace summary** menu. To put a highlight on a class, you can either:

- define it the following way:

```
class ** Example
```

- or declare all the highlighted classes *after the definition section*, with the following syntax:

```
highlight ClassId1 ClassId2 ... ClassIdN
```

You may also want to record the list of trace summaries associated with each TAG tree. To do so, just use the **-t** option followed by a filename when invoking XMG:

```
MetaTAG metagrammar.mg -t traceSummaries.txt
```

As a result the traceSummaries.txt file will look like the following:

```
Famille : Tree
Solution : TTree-1
Trace summary :
        SujCan
```

6.6 Defining mutexes

Another functionality provided by the XMG system is the possibility to define **mutexes**. This means that you can specify which classes are incompatible. So you can prevent some class combinations. To do so, you first have to define a mutual exclusion set by typing **mutex *Id*** such as in:

```
mutex SUBJ-INV
```

Then you add classes in this set by invoking **mutex *Id* += *ClassId*** such as in:

```
mutex SUBJ-INV += CanonicalObject
mutex SUBJ-INV += InvertedNominalSubject
```

Here we specify that we cannot use in the same description both the `CanonicalObject` and the `InvertedNominalSubject` classes.

N.B.: note that in the metagrammar file, the mutex definitions have to be placed after the type, property and feature declarations and before the value instructions.

6.7 Using parameterized classes

We have seen previously that it was possible to use the *interface* (ie dynamic dimension) to share information between classes. For instance we see in the Predicate Semantics section, that we can define a semantic class the following way:

```
class BinaryRel
declare !L ?X ?Y ?P
{ <sem>{!L:?P(?X,?Y)}
*= [pred=?P] }
```

thus we used the interface to specify that the value of the predicate will be given when the class will be combined. To define the predicate we will reuse the **pred** feature in the interface (in which features have a global scope), and associate to it a constant value, such as in:

```
class lexemeManger
{ transitive[] ; BinaryRel[]*=[pred=manger] }
```

(note the constant `manger` have to be declared in the header of the metagrammar)

XMG also offers another way of performing such information sharing between classes: *using class parameters*. We can define a parameterized class by defining the class with the following syntax: **class *Id* [*p1, ... ,pN*]**. Note that the *p1, ... ,pN* identifiers do not have to be declared, and can be used in the content of the class. Later when the class is called, you can give a value to its parameters. Back to our semantic example, another way of giving a value to the predicate would be:

- first to define the `BinaryRel` as a parameterized class:

```
class BinaryRel[P]
declare !L ?X ?Y
{ <sem>{!L:?P(?X,?Y)} }
```

- and second to instanciate the predicate when the class is called:

```
class lexemeManger
{ transitive[] ; BinaryRel[manger] }
```

(note that we still have to define `manger` in the header)

6.8 Restricted import

In the Import subsection, we have seen how to use inheritence to reuse the content of a class and specialize it. Actually the import instruction comes with alternative uses. More precisely we do not have to import all the identifiers that are exported by a mother class. For instance one would like to *restrict* the import to some specific identifiers (to avoid name conflicts). This can be done by typing **import *ClassId[]* as [*X1, ..., Xn*]**. For instance we can redefine the `transitive` class introduced above the following way:

```
class transitive
import Object[] as [?X,?Y]
declare ?SU ?Z
{
        { ?SU=CanSubject[] | ?SU=RelSubject[] } ; ?Z = Active[] ;
        ?SU.?X = ?X ; ?SU.?Z = ?Y ;
        ?SU.?X = ?Z.?X ; ?SU.?Z = ?Z.?Y
}
```

Now we make the `transitive` class inherit from `Object` and *restrict the import to the ?X and ?Y variables* (since the ?Z variable representing the object complementizer is not needed here). Thus (1) we do not need to use an ?OB variable to refer to the exported identifiers of the `Object` class any more, and (2) we can use the ?Z variable to refer to the exported identifiers of the `Active` class without any name conflict.

6.9 Import with identifier renaming

Another way of avoiding name conflicts when specializing a class is to use *identifier renaming*. This can be done by using the alternative import declaration: **import *ClassId[]* as [...,*Xi*=*Yi*,..]**. Back to the `transitive` example of the preceding section, we can redefine it the following way:

```
class transitive
import Object[] as [?X=?A,?Y=?B]
declare ?X ?Z
{
        { ?X=CanSubject[] | ?X=RelSubject[] } ; ?Z = Active[] ;
        ?X.?X = ?A ; ?X.?Z = ?B ;
        ?X.?X = ?Z.?X ;  ?X.?Z = ?Z.?Y
}
```

Here we still make the `transitive` class inherit from the `Object` class, but with *renaming of Object's exported identifiers* so that the ?X variable is now called ?A and the ?Y variable ?B. As a consequence, we can use a local ?X identifier (now referring to the record of `CanSubject`'s or `RelSubject`'s exported identifier) without raising a name conflict. In the node unification equations, we use ?X instead of ?SU previously, and ?A (resp. ?B) instead of ?X (resp. ?B) in the preceding section.

6.10 Giving a global name to a node

To perform interfacing with the lexicon, one may want to give global names to some specific nodes. So one will be able to refer to these nodes in the lexicon. Such an interfacing can be used for instance to manage semantic information. To associate global names *that will appear* in the semi-automatically produced grammar, you have to:
1. declare an *enumerate type* containing all the names you will use:

```
type NAME = {subjNode, objNode, anchor}
```

2. declare a property *name* of this type:

```
property name  : NAME
```

3. associate to the specific nodes the predefined names:

```
node (mark=subst,name=objNode)[cat=n]
```

N.B.: make sure these name properties will not cause node unification failures, ie do not give different names to nodes that will be merged.
At the end, the node name are displayed on screen, and in the XML file (as an attribute of the node element):

```
<node type="subst" name="objNode">
```

6.11 Declaring semantic classes

To support semantic instanciation when anchoring a grammar with a lexicon, we can extract the semantic information from the metagrammar. This can be done by declaring whose classes contain relevant semantic information.

```
semantics ClassId1 ClassId2 ... ClassIdN
```

(note that the semantic classes are declared after the definition section)

Thus, when using the **--mac** XMG option, the compiler will extract the semantic information contained in these classes and produce a file containing semantic *macros* (these can be used within the lexicon). Use *-s* to specify the name of the file containing these macros (the *.mac* extension will be automatically added). These macros will associate names (class names) with semantic information (class contents).

Note that these semantic classes have specific features in their interface. These features correspond to pieces of information coming from the lexicon, *i.e.* these are kind of semantic macro arguments. So provided we can specifiy what features are instantiated according to the lexicon, we can in practice extract *real* semantic macros from the metagrammar. These arguments are declared as external features:

```
extern feat1 feat2 ... featN
```

N.B. the external features have to be declared within the definition section (*i.e.* before defining the classes).

Thus XMG will produce parameterized macros, whose body will consist of semantic class content and interface.

Another interest of declaring semantic classes is to extract *links* between valuation classes and semantic classes: which semantic classes are used by a given valuation ? This can be extracted from the metagrammar by using XMG's --*lin* option. This will create a file containing these relations. Use the *-l* option to specify the name of the file containing these links (the *.lin* extension will be automatically added).

6.12 XMG’s graphical output interface

As we have seen, XMG offers a graphical ouput interface representing TAG trees. Some additional explanations may be useful to understand all the information that is displayed in the syntactic structure (see XMG’s ouput for the description of the other pieces of information displayed such as the trace, etc).
Typically a TAG tree in XMG looks like:

TAG Tree

That is, the syntactic tree corresponds to a tree structure whose nodes are equipped with a complex feature-structure (FS). This FS usually contains 3 features: **cat** for the syntactic category, **top** and **bot** for the top and bottom feature-structures of the TAG formalism. On top of that, a node may have been given a **`mark`** (as a node property):

```
type MARK = {subst, subst, nadj, foot, anchor, coanchor, flex }
property mark : MARK
... node ?Y (mark = subst, gf=subj)[cat=n] ...
```

Such a property has a special status within the XMG system. We said previously that properties are used either to perform additional treatments on syntactic structures or to decorate nodes. The `mark` property is of the second type, it is recognized by XMG so that it appears in both the graphical output (Graphical User Interface - GUI) and the XML file. In the tree above, we have two nodes marked as substitution nodes, this is indicated by the exclamation mark printed in red on the top left hand corner of the node. The special marks are given in the following table:

Value of the mark property	**Symbol appearing in the GUI**	**Signification**
subst	!	Substitution node
nadj	#	Null adjunction node
foot	*	Foot node
anchor	<>	Anchor node
coanchor	<>2	Coanchor node
flex	<->	Lexical item in the tree schemata

The other property that appears in the GUI is the node name. It is displayed in red on the bottom right hand corner of the node:

TAG tree with named nodes

6.13 Printing trees in postscript files

It is recommended to use tools for capturing screenshots. Note that since version 1.1.7 you can define the features to be displayed on screen.

6.14 Designing metagrammars with emacs

To ease metagrammar development, a Metagrammar mode for emacs has been written. To install it, just copy the lines below in your emacs' configuration file (usually ~/.emacs):

```
;;=====================================================================
;; MetaGrammar mode
;; Mode used to design MetaGrammars avec emacs (highlight)
;; Author: B. Crabbe

(require 'generic-x) ;;pour Emacs OK, mais semble ne pas marcher avec XEmacs
(define-generic-mode 'metagrammar-mode
  ;;comments
  '("%" "//")
  ;;keywords
  '("class" "node" "value" "syn" "sem" "semantics" "highlight" "extern" "feature" "type" "property")
  '(
    ;;noms de classes
    ("class\\s (\*\*)* +\\(\\sw[a-zA-Z0-9_.-]*\\)" 1 'font-lock-type-face)
    ("\?[a-zA-Z0-9]+" . font-lock-variable-name-face)
    ("\![a-zA-Z0-9]+" . font-lock-constant-face)
    ;;params & node props
    ("\(\\(\\sw[a-zA-Z0-9_.-]*\\(,\\sw[a-zA-Z0-9_.-]*\\)*\\)\)" 1 font-lock-constant-face)
    ;;params inside
    ("$\\(\\sw*\\)*" . font-lock-constant-face)
    )
  ;;file extension
  '(".mg\\'")
  nil
  "Major mode for metagrammar editing")
;;=====================================================================
```

As a result, when the emacs highlighting option will be used, specific information will be displayed with specific colors:

Metagrammar in emacs

N.B.: there also exists a *vim* mode available at http://sourcesup.cru.fr/xmg/xmg.vim.

6.15 List of XMG's options

The XMG system provides 2 executable programs, namely **`MetaTAG`** and **`MetaIG`**. Both of these require as an argument the name of the main metagrammar file (ie this file may *include* other files). So the basic usage is:

```
MetaTAG metagrammar_file
```

to compile a TAG metagrammar to produce a TAG, or

```
MetaIG metagrammar_file
```

to compile an IG metagrammar to produce an IG.

6.15.1 MetaTAG

The **MetaTAG** executable can be used with one of the following options (--gui being the default):

```
--gui | --xml | --chk | --all | --mac | --lin
```

--gui	to print the TAG trees through a Qtk GUI (default)
--xml	to print the TAG trees in an XML format. By default this XML code is printed on stdout, but you can specify a file name by typing **`-o file.xml`**
--chk	to print the TAG trees in a binary format (Pickle) for using with the XMG tools. You need to specify the output file by typing **`-c file.rec`**
--all	to print the grammar both in an xml file and in a binary file. You can specify the output files with the **`-o`** and **`-c`** options for the xml file and binary file respectively
--mac	to extract semantic macros from classes declared as semantics (see Declaring semantic classes). You need to specify the output file by typing **`-s file`** (the .mac extension will be added).
--lin	to extract links between valuations and semantic classes. You need to specify the output file by typing **`-l file`** (the .lin extension will be added).

Besides it can be used with additional options:

```
-t traceSummaries.txt
```

to print the trace summaries in a text file (see Using trace summaries),

```
-v
```

to activate a verbose mode (information about the compilation, *eg* warnings from the XMG compiler),

```
-d <DTD_PATH>
```

to check the validity of the XML produced (you need to have xmllint installed).

```
-m <Integer>
```

to split the grammar in several xml files (only compatible with --xml), m stands for *many*.

6.15.2 MetaIG

The **MetaIG'**s options are the following:

```
--xml
```

to prints the IG tree descriptions in an XML format in stdout (default). Use the **`-o`** option to specify the name of the XML output file. Note that a GUI to visualize the d-trees of the Interaction Grammar is available within the LEOPAR system, see LEOPAR,

```
-v
```

to activate a verbose mode (information about the compilation, *eg* warnings from the XMG compiler),

```
-d <DTD_PATH>
```

to check the validity of the XML produced (you need to have xmllint installed).

7 Caveats

In this section, we give some information about principles that are applied by the compiler and that can cause unexpected results.

7.1 Island principle

If an elementary tree contains both a node with property (*extracted* = +) and a node marked as a *completive*, then the latter has to be a substitution node. If it contains only a node marked as completive (*i.e.* no extracted node), then this completive node has to be a foot node.

Such a principle can remove trees from the grammar or add a mark (subst or foot) to certain nodes **without any warning**.

For a detailed presentation of the principle, please see [Crabbé, 05b] page 182.

7.2 TAG validity principle

At the end of the tree description solving process, if there are trees containing either:

- inner nodes marked differently as std (*i.e.* standard, no mark) or nadj (*i.e.* non-adjunction)
- 2 or more nodes marked as foot

Then these trees will be removed with the printing of a **warning message** on stdout.

N.B.

- Note that these 2 principles represent linguistic properties that have to be respected within the TAG grammar. These principles cannot be deactivated when using `MetaTAG`.
- See also the XMG *Common pitfalls wiki page* at http://wiki.loria.fr/wiki/XMG/Common_pitfalls.

8 Conclusion and Future Work

We have presented here a new metagrammatical framework that can supports several grammatical formalisms (TAG and IG at the moment).

The XMG system is freely available at http://sourcesup.cru.fr/xmg under the terms of the CeCILL license. It has been developed in Oz/Mozart. The supported platforms are Linux / Unix, Mac and Windows.

XMG has been used successfully to develop a wide coverage TAG for French (see [Crabbé, 05a] and [Gardent and Parmentier, 05]) and a medium size IG. This TAG metagrammar, containing 285 classes, produces about 5,000 *non-anchored* TAG trees. It has been evaluated in syntactic parsing on the TSNLP. The results are encouraging since the success rate is about 75% (see [Crabbé, 05b]). To give an overview of the efficiency of the system, it takes 5 minutes to compile this grammar on a Pentium 4 - 2.66 Ghz processor with 1 Go RAM.

We plan to develop a library of dimensions, each equipped with a specific language. This will allow the description of an arbitrary number of grammatical formalisms by using adequate dimensions.

We are also working on the use of the automatically produced wide coverage TAG with semantic information in the context of parsing [Gardent and Parmentier, 05] and generation [Gardent and Kow, 04].

9 Acknowledgements

We are particularly grateful to Guillaume Bonfante, Eric De La Clergerie, Benoît Crabbé, Denys Duchier, Bertrand Gaiffe, Claire Gardent, Bruno Guillaume, Eric Kow, Guy Perrier, Sylvain Pogodalla, and Azim Roussanaly for their useful comments, ideas, pieces of advise in the context of this work. We would like to thank also the members of the *Calligramme* and *Langue et Dialogue* projects at LORIA for profitable discussions. In particular, we would like to thanks Sébastien Hinderer for his reviewing.

Thanks also to Jan Kärrman from the Dept. of Information Technology of the Uppsala University - Sweden for developing the html2ps tool.

10 References

In this section, we do not give an exhaustive bibliography, we rather give the references of the papers that are relatively closely linked to this work.

[Ait-Kaci, 91]	Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*, in proceedings of the Eighth International Conference in Logic Programming, MIT Press, 1991.
[Bonfante et al., 03]	Guillaume Bonfante, Bruno Guillaume and Guy Perrier. *Analyse syntaxique électrostatique* in Evolutions en analyse syntaxique, Revue TAL (Traitement Automatique des Langues), volume 44:3, 2003.
[Candito, 99]	Marie-Hélène Candito. *Représentation modulaire et paramétrable de grammaires électroniques lexicalisées : application au français et à l'italien*, Université Paris 7, 1999.
[Crabbé et al., 04a]	Benoît Crabbé, Bertrand Gaiffe and Azim Roussanaly. *Représentation et gestion du lexique d'une grammaire d'arbres adjoints* in Traitement Automatique des Langues, 43,3, 2004.
[Crabbé and Duchier, 04b]	Benoît Crabbé and Denys Duchier. *Metagrammar Redux* in Proceedings of CSLP'04, Roskilde, Denmark, 2004.
[Crabbé, 05a]	Benoit Crabbé. *Grammatical development with XMG*, in proceedings of the Fifth International Conference on Logical Aspects of Computational Linguistics (LACL05), Bordeaux, 2005.

[Crabbé, 05b] Benoit Crabbé. *Représentation informatique de grammaires fortement lexicalisées - Application à la grammaire d'arbres adjoints*, Université Nancy 2, 2005.

[Debusmann et al., 04] Ralph Debusmann, Denys Duchier and Geert-Jan. M. Kruijff. *Extensible Dependency Grammar: A New Methodology*, in proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar, Geneva, 2004.

[Duchier and Niehren, 00] Denys Duchier and Joachim Niehren. *Dominance Constraints with Set Operators*, in proceedings of the First International Conference on Computational Logic (CL2000), volume 1861 of the Lecture Notes in Computer Science, pages 326-341, Springer.

[Duchier, 00] Denys Duchier. *Constraint Programming For Natural Language Processing*, Lecture Notes, ESSLLI 2000. Available at http://www.ps.uni-sb.de/Papers/abstracts/duchier-esslli2000.html, 2000.

[Duchier et al., 04] Denys Duchier, Joseph Le Roux and Yannick Parmentier. *The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture*, in proceedings of the Second International Mozart/Oz Conference (MOZ'2004), Charleroi, 2004.

[Gardent and Kallmeyer, 03] Claire Gardent and Laura Kallmeyer. *Semantic construction in FTAG*, in proceedings of the 10th meeting of the European Chapter of the Association for Computational Linguistics, Budapest, 2003.

[Gardent and Kow, 04] Claire Gardent and Eric Kow. *Génération et sélection de paraphrases grammaticales*, journée ATALA sur la génération de Langue Naturelle, Paris, 2004.

[Gardent and Parmentier, 05] Claire Gardent and Yannick Parmentier, *Large scale semantic construction for Tree Adjoining Grammars*, In proceedings of the Fifth International Conference on Logical Aspects of Computational Linguistics (LACL05), Bordeaux, 2005.

[Joshi and Shabes, 97] Aravind Joshi and Yves Schabes. *Tree-Adjoining Grammars*, in Handbook of Formal Languages, G. Rozenberg and A. Salomaa editors, Springer, Berlin, New York, volume 3, pages 69 - 124, 1997.

[Pereira and Warren, 1980] Fernando Pereira and David H. D. Warren. *Definite clause grammars for language analysis ---A survey of*

the formalism and a comparison to augmented transition networks, *Artificial Intelligence, volume 13, 1980.*

[Perrier, 03] Guy Perrier. *Les grammaires d'interaction*, Habilitation à diriger les recherches en informatique, Université Nancy 2, 2003.

[Rogers and Vijay-Shanker, 94]	James Rogers and Vijay K. Shanker. *Obtaining trees from their descriptions: An application to tree-adjoining grammars*, Computational Intelligence, 10:401--421, 1994.
[Thomasset and De La Clergerie, 05]	François Thomasset and Eric Villemonte de la Clergerie. *Comment obtenir plus des Méta-Grammaires*, in proceedings of TALN'05, Dourdan, France, 2005.
[Van Roy, 90]	Peter Van Roy. *Extended DCG Notation: A Tool for Applicative Programming in Prolog*, Technical Report UCB/CSD 90/583, Computer Science Division, UC Berkeley, 1990.
[Vijay-Shanker and Shabes, 92]	Vijay K. Shanker and Yves Schabes. *Structure Sharing in Lexicalized Tree Adjoining Grammars*, in proceedings of the 16th International Conference on Computational Linguistics (COLING'92), Nantes, pp. 205 - 212, 1992.
[Xia et al., 99]	Fei Xia, Martha Palmer and Vijay K. Shanker. *Toward semi-automating grammar development*, in proceedings of the 5th Natural Language Processing Pacific Rim Symposium(NLPRS-99), Beijing, China, 1999.
[XTAG group, 01]	XTAG-Research-Group. *A Lexicalized Tree Adjoining Grammar for English*, IRCS, University of Pennsylvania, IRCS-01-03, 2001.

11 Contacts

joseph.leroux@loria.fr
yannick.parmentier@loria.fr

Yparmenti 1 sep 2005 à 18:08 (CEST)

Outils personels

- Yparmenti
- Ma page de discussion
- Préférences
- Liste de suivi
- Mes contributions
- Déconnexion

Navigation

- Accueil
- Modifications récentes
- Une page au hasard
- Aide

Rechercher

Boîte à outils

- Pages liées
- Suivi des liens
- Copier sur le serveur
- Pages spéciales
- Version imprimable

MediaWiki

- Dernière modification de cette page le 7 déc 2006 à 14:51.
- Cette page a été consultée 25780 fois.
- À propos de Loria Wiki
- Avertissements