Grammar Implementation: XMG

XMG Tutorial

Laura Kallmeyer & Benjamin Burkhardt
(Slides partly by Timm Lichte and Simon Petitjean)

HHU Düsseldorf

WS 2017/2018

How does it work?

XMG processing steps are as follow:

- The metagrammar is compiled: metagrammatical language is translated into executable code
- The generated code is executed: accumulation of descriptions into the dimensions
- Descriptions are solved: every dimension comes with a dedicated solver
- Models are converted into the output language (XML)

Tools

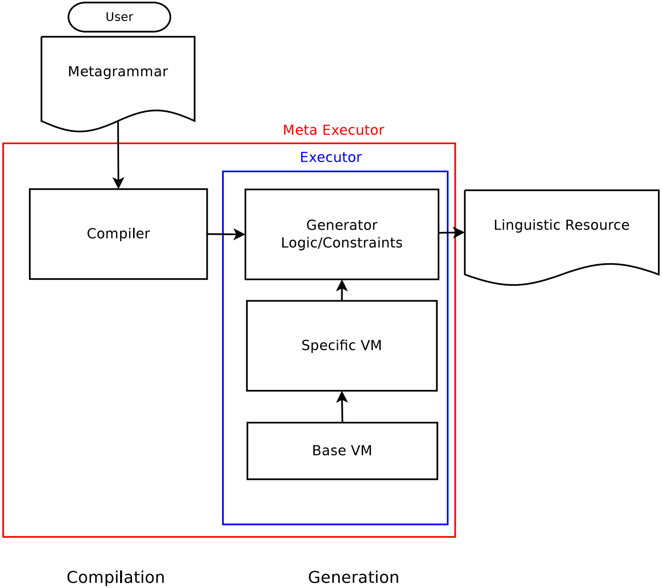
XMG-1

- eXtensible (?) Metagrammar
- Only 3 dimensions

XMG-2

- Arbitrarily many dimensions, with DSLs
- Modular assembly of DSL, using bricks
- Methodology to generate a whole processing chain

XMG-2: Architecture (relevant part for us)



Installing XMG 2

Three options, provided by the documentation:
`dokufarm.phil.hhu.de/xmg`

- Follow the steps (Ubuntu), or
- Install VirtualBox and get the XMG image
- Use the online compiler(s): `http://xmg.phil.hhu.de/index.php/upload/compile_grammar`

Installing contributions

- Making a contribution available is done with the `install` command

```
xmg@xmg:~/xmg-ng$ cd contributions
xmg@xmg:~/xmg-ng/contributions$ xmg install core
xmg@xmg:~/xmg-ng/contributions$ xmg install treemg
xmg@xmg:~/xmg-ng/contributions$ xmg install compat
xmg@xmg:~/xmg-ng/contributions$ xmg install
    synsemCompiler
```

Installing compilers

- A set of already assembled compilers is available
- Building one of them can be done with the `build` command

```
xmg@xmg:~/xmg-ng$ cd contributions/synsemCompiler/
xmg@xmg:~/xmg-ng/.../synsemCompiler$ cd compilers/
    synsem/
xmg@xmg:~/xmg-ng/.../synsem$ xmg build
```

- To avoid these steps: scripts (`reinstall.sh`)

Compiling a first metagrammar

The `compile` command takes two arguments

- The compiler which will be used
- The metagrammar

```
xmg@xmg:~/xmg-ng$ xmg compile synsem MetaGrammars/synsem
/TagExample.mg
```

Drawing trees

The output of XMG2 can be given to a parser or a generator, but also be inspected by a tree viewer

- XMG comes with a built-in tree viewer:

```
xmg@xmg:~/xmg-ng$ xmg gui tag
```

- Pytreeview (`https://gitlab.com/parmenti/pytreeview`) is a light tree viewer installed on the Virtualbox distribution of XMG2:

```
xmg@xmg:~/xmg-ng$ pytreeview --mode WEB -i input-file.
    xml
```

- A tree and frame viewer is available online: `http://xmg.phil.hhu.de/index.php/upload/xmg_viewer`

The control language

XMG descriptions:

- Associate a content to an identifier (abstraction)
- Describe structures inside dimensions, with dedicated languages
- Use other abstractions (classes)
- Combine contents in a disjunctive or a conjunctive way

$$\begin{aligned} Class &:= \quad Name \longrightarrow Content \\ Content &:= \quad \langle Dimension \rangle \{Description\} \ \mid \ Name \ \mid \\ & \quad\;\; Content \vee Content \ \mid \ Content \wedge Content \end{aligned}$$

Describing trees

The <syn> dimension

- Declaring nodes: keyword **node**, optional node variable, optional features and properties
  node ?S [cat=s]
- Expressing constraints between nodes: dominance operators (**->**, **->+**, **->***) and precedence operators (**>>**, **>>+**, **>>***)
- Combining these statements: with logical operators (**;** and **|**)

Example:

```
node ?S [cat=s];
node ?VP [cat=vp];
node ?V (mark=anchor) [cat=v];
node ?NP (mark=subst) [cat=n];
?S -> ?VP;
?VP -> ?V;
?S -> ?NP;
?NP >> ?VP
```

Alternative syntax: bracket notation

The <syn> dimension

- Declaring nodes: same as for the standard notation
- Expressing dominance and precedence constraints thanks to bracketing, and special operators for non immediate relations

```
node ?S [cat=s]{
  node ?NP (mark=subst) [cat=np]
  node ?VP [cat=vp]{
    node ?V (mark=anchor) [cat=v]
  }
}
```

Using dimensions

Contributing descriptions

- Descriptions (constraints) are accumulated into dimensions
- Every dimension is associated to a solver (sometimes identity)
- **<syn>**: a tree solver generates all minimal models

```
<syn>{
    node ?S [cat=s];
    node ?VP [cat=vp];
    node ?V (mark=anchor) [cat=v];
    node ?NP (mark=subst) [cat=n];
    ?S -> ?VP;
    ?VP -> ?V;
    ?S -> ?NP;
    ?NP >> ?VP
}
```

Syntactic nodes

Two nodes can be unified if:

- their feature structures can be unified
- their properties can be unified

Unification of nodes happens at two different stages:

- During the execution of the code ("explicit" unification: unification instruction `=` or reuse of variable)
- After solving: some nodes may be merged to obtain a minimal model

Minimal models

A minimal model is a model of the description where:

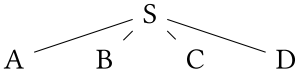
- no constraint is violated
- no additional node is created

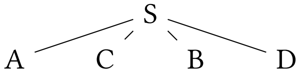
What are the minimal models for the following sets of constraints?

```
?S ->+ ?A ; ?S -> ?B
```

```
?S -> ?A ; ?S -> ?B ; ?S -> ?C ; ?A >>* ?C
```

Which set of constraints leads to the following minimal models?





Definition of types and constants

Everything inside the metagrammar has a type: values, feature structures, nodes, dimensions…

Four ways to define new types:

- Enumerated type: type T={a,b,c,d}
- Structured type: type T=[a_1:t_1,…,a_n:t_n]
- Interval type: type T=[1..3]
- Unspecified type: type T!

Definition of types and constants

We can now specify the types of features and properties:

```
type CAT=  {np,vp,s,n,v,det}
type MARK= {lex,anchor,subst}
type LABEL !
type PERS= [1..3]
type GEN = {m,f}
type NUM = {sg,pl}
type AGR = [gen:GEN, num:NUM]


feature cat:  CAT
feature e:    LABEL
feature pers: PERS
feature agr:  AGR

property mark: MARK
```

Principles: motivation

- As fragments become more numerous, controlling their combination (and the scope of variables) gets difficult
- Idea: adding new constraints on top of dominance and precedence
- Principles: sets of additionnal constraints for the solver[CrabbeDuchier:04]

A set of principles

XMG offers several sets of additionnal constraints over the models (principles):

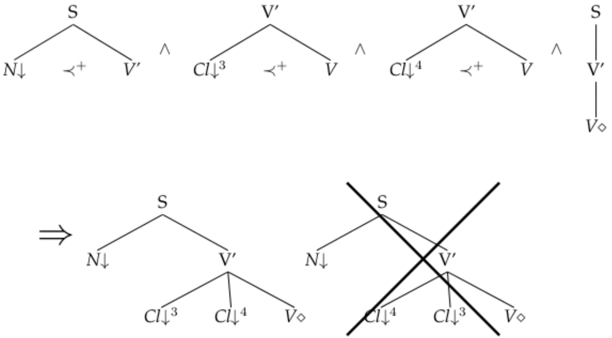
- colors: polarities for node unification
- rank: linear order constraints on nodes
- unicity: uniqueness of a feature inside a model

Rank: Clitics ordering

- The ordering of clitic pronouns (in Spanish or French for example) is known to be problematic when formalizing a grammar
- In a metagrammar, when combining fragments, nodes representing these clitics have to come in a specific order

- Pedro nos la da
- *Pedro la nos da
- Je le lui laisse
- *Je lui le laisse

Rank: Clitics ordering (in French)



- Every produced model has to satisfy the order constraint

Using principles: rank

```
use rank with () dims (syn)
type RANK=[1..7]
property rank: RANK
```

```
class CliticIobjectII
import nonReflexiveClitic[]
{
  <syn>{
    node xCl(rank=2)
            [top=[func=iobj, pers = @{1,2}]]
  }
}
```

Using principles: unicity

```
use unicity with (rank=1) dims (syn)
use unicity with (rank=2) dims (syn)
use unicity with (rank=3) dims (syn)
use unicity with (rank=4) dims (syn)
use unicity with (rank=5) dims (syn)
use unicity with (rank=6) dims (syn)
use unicity with (rank=7) dims (syn)
```

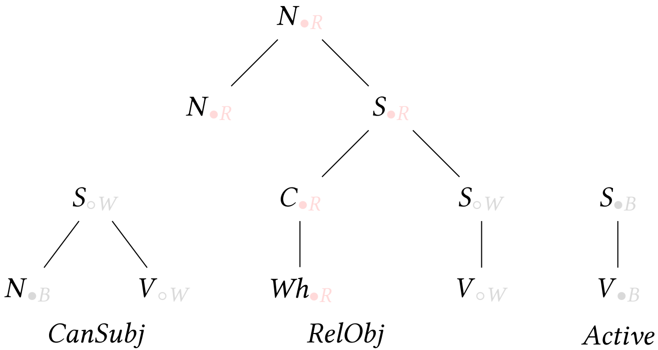
Using principles: colors

- Colors are a solution to guide the combination of fragments
- A color is affected to every node
- New constraints on node unification

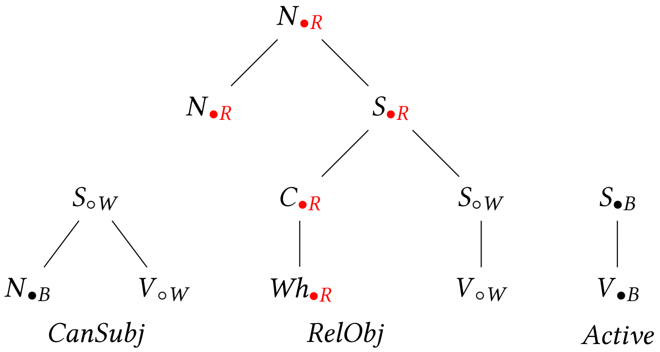
	$\bullet_B$	$\bullet_R$	$\circ_W$	$\perp$
$\bullet_B$	$\perp$	$\perp$	$\bullet_B$	$\perp$
$\bullet_R$	$\perp$	$\perp$	$\perp$	$\perp$
$\circ_W$	$\bullet_B$	$\perp$	$\circ_W$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

- Valid models only have red and black nodes

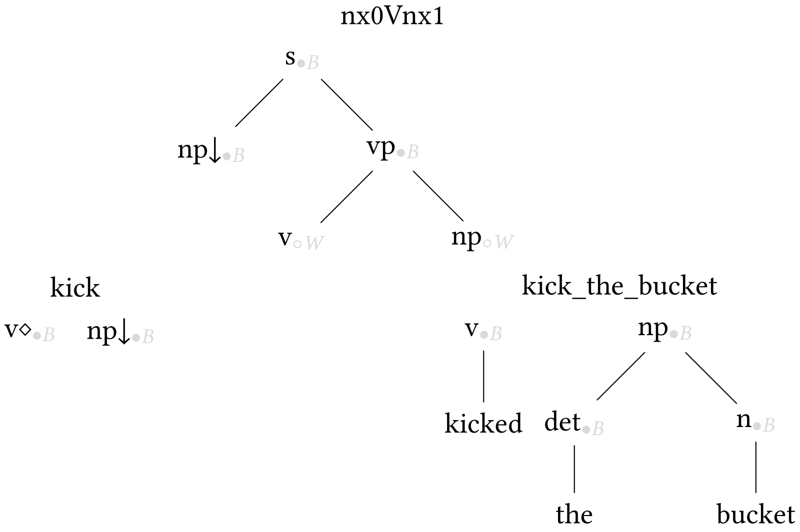
Combination with polarities



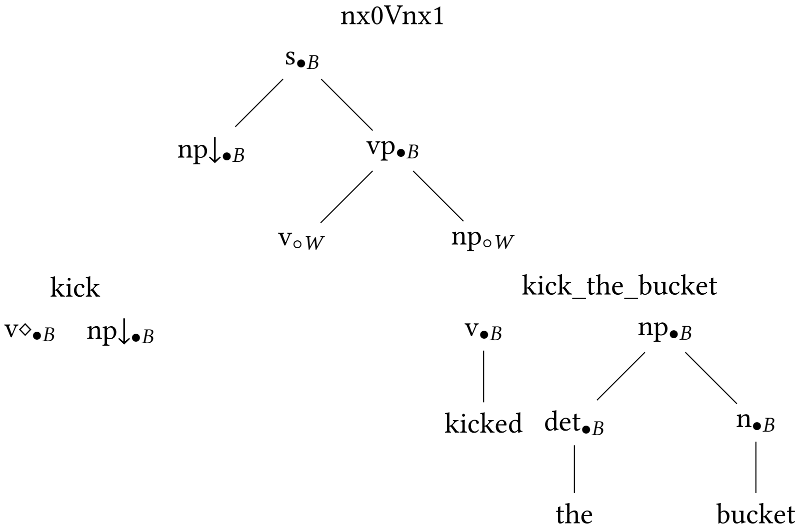
Combination with polarities



Combination with polarities



Combination with polarities



Using principles: colors

```
use color with () dims (syn)
type COLOR={red,black,white}
property color: COLOR
```

```
class nx0Vnx1
declare ?S ?NP_Subj ?VP ?V ?NP_Obj
{
  <syn>{
      ?S (color=red)[cat=s] {
        ?NP_Subj (color=black, mark=subst) [cat=np]
        ?VP (color=black)[cat=vp] {
          ?V (color=white)[cat=v]
          ?NP_Obj (color=white)[cat=np]
        }
      }
  }
}
```