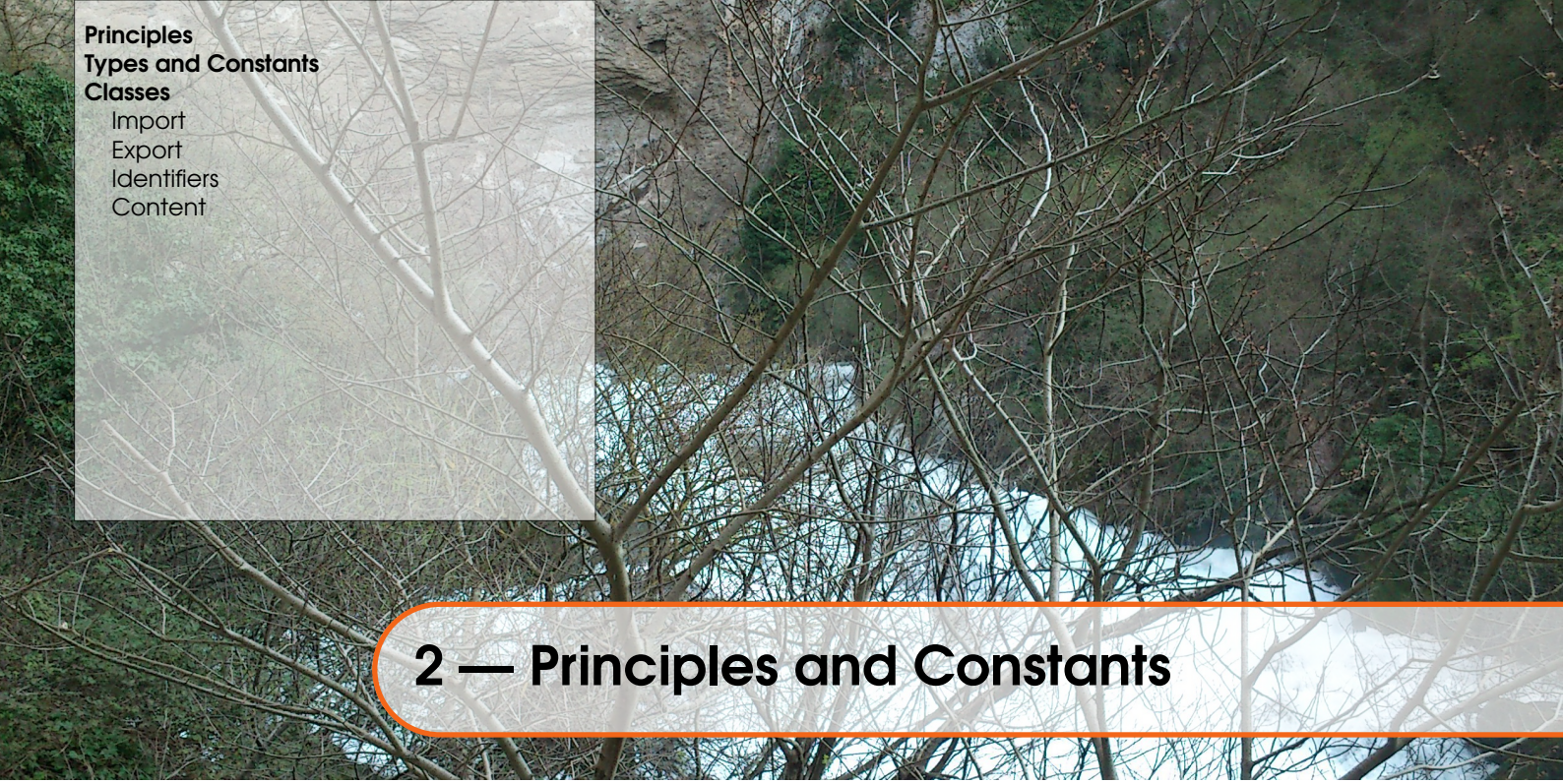
Writing a XMG MetaGrammar

Simon Petitjean

Contents

1 — Introduction

About XMG MetaGrammars.



2 — Principles and Constants

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).

2.2 Types and Constants

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.

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.

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.

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)

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.

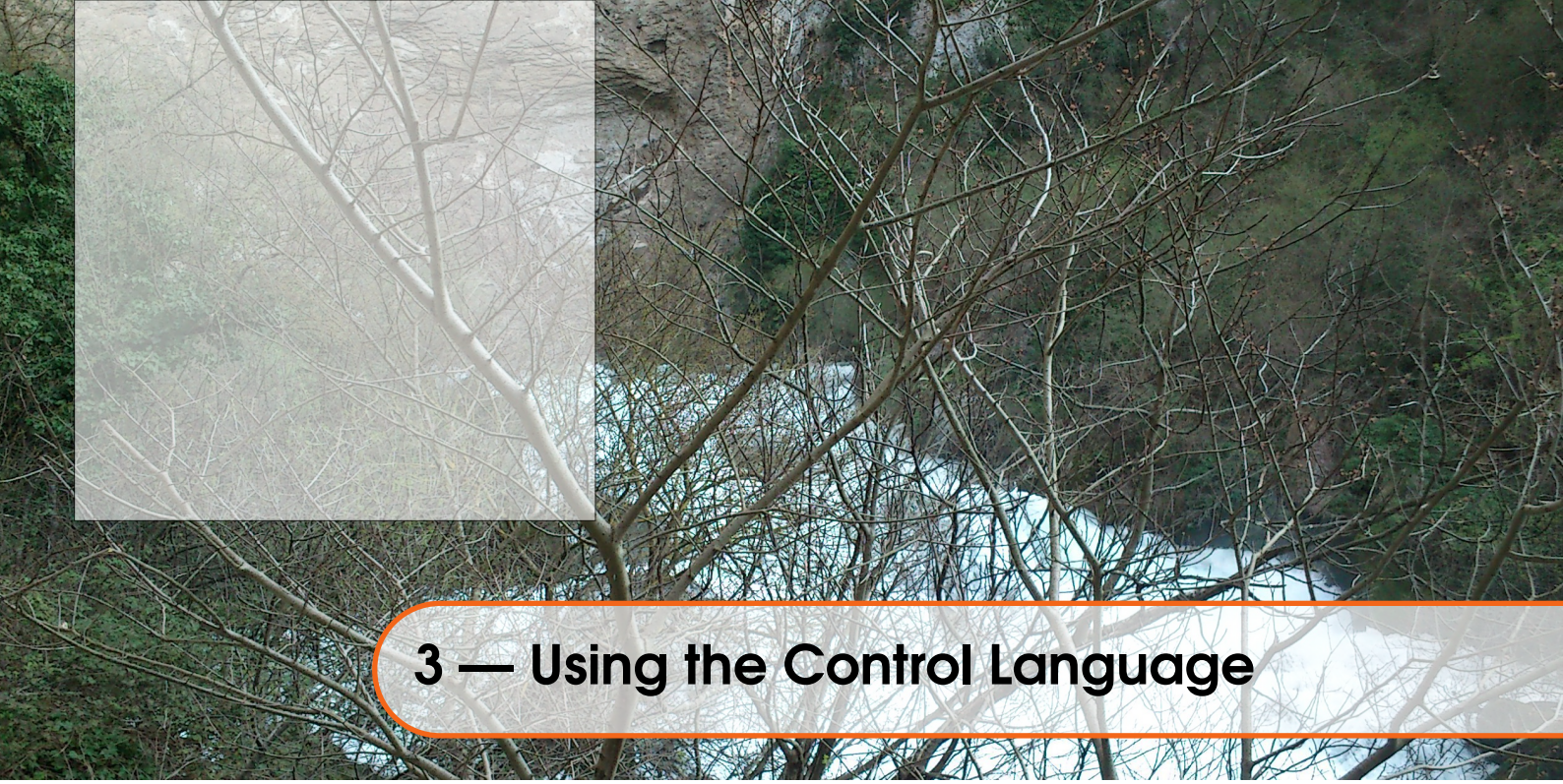
2.3.4 Content

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"
- a statement associated to an interface (see below)

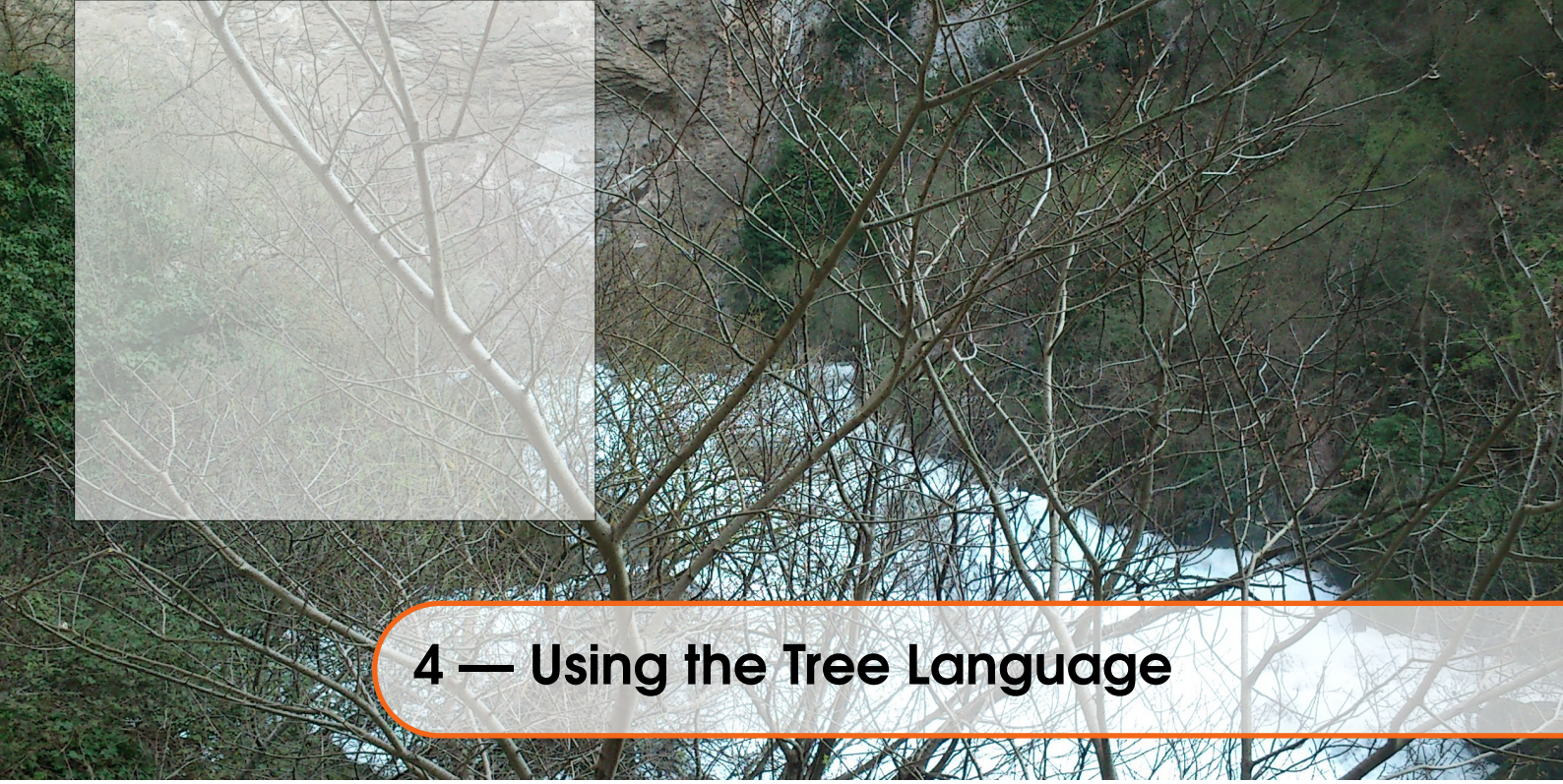
By statement we mean:

- 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),
- a description belonging to a dimension



3 — Using the Control Language

For Calls, Conjunctions and Disjunctions



4 — Using the Tree Language

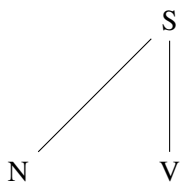
A syntactic description is given following the pattern <syn> formulas . Now what kind of formulas doas a syntactic description contain ? The answer is nodes. These nodes are in relation with each other. In XMG, you may give a name to a node by using a variable, and also associate properties and/or features 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:



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.

About predicates

To describe fields and features

Books

Articles

C

D

E

F

L

N

P

R

T

V