Domain Specific Language Design (2IMP20)

Textual Concrete Syntax: Xtext

Loek Cleophas, Ivan Kurtev

Agenda

Model-based way of DSL definition

Textual concrete syntax

- Xtext

Overview of Language Aspects

Every language has a number of ingredients (aspects)

Abstract Syntax

- Metamodel

Concrete Syntax

- Textual
- Visual

Static Semantics

- Validity constraints
- Type rules
- Scoping

Dynamic Semantics

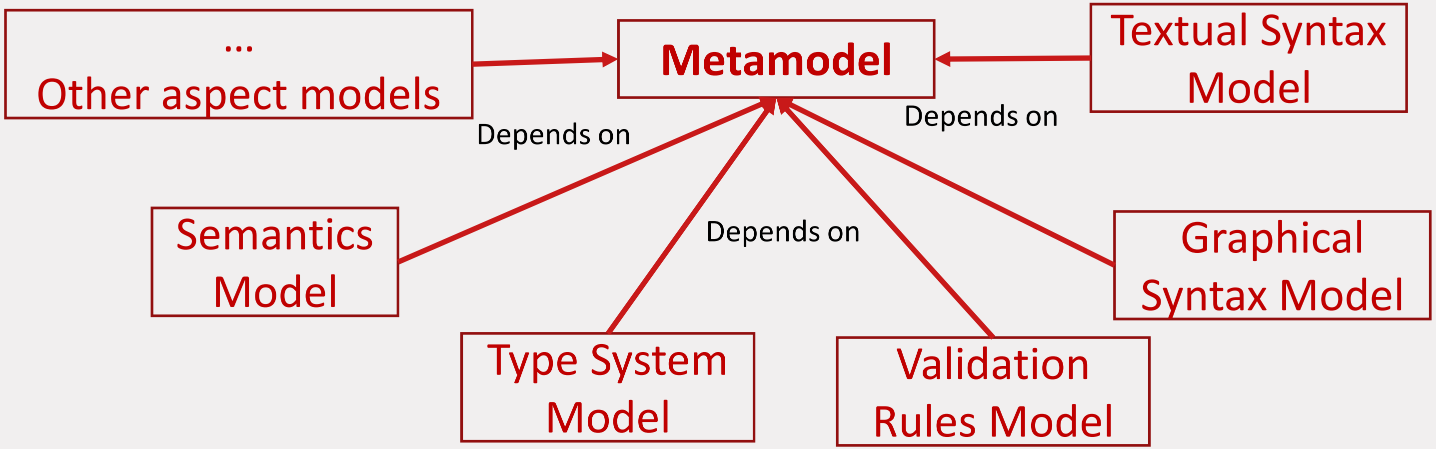
Others:

- Generators
- Analysis tools
- Interpreters
- Simulators
- Debuggers

Until now we have covered *Abstract Syntax* (by means of a *metamodel*), *Validity constraints* (with *OCL*), and generation (with model transformations)

TU/e

Model-based DSL Definitions



In Modelware, a DSL is defined as *a set of models* that specify the relevant language aspects. The metamodel plays a central role. Not all language aspects need to be always present.

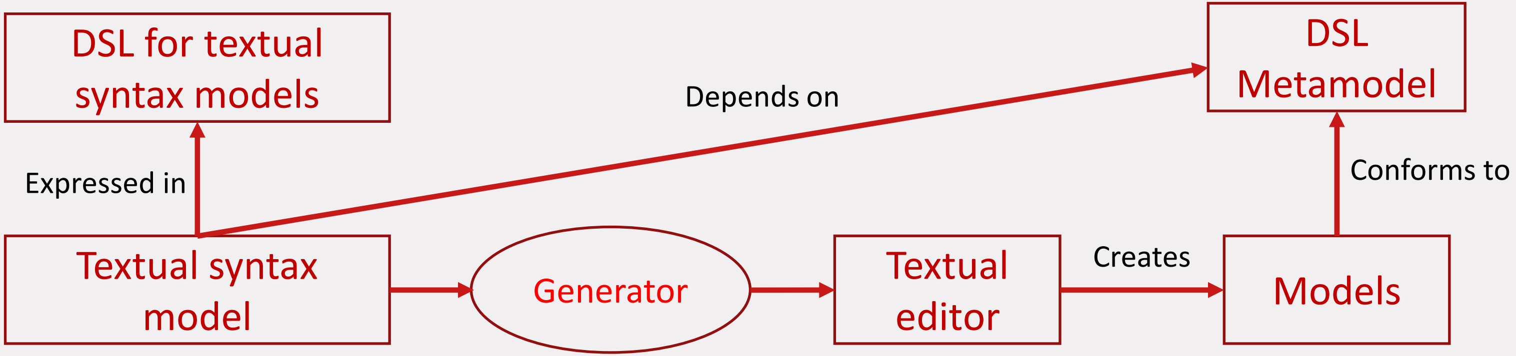
Model-based DSL Definition

Models of language aspects are often expressed in DSLs

- Example: OCL (presented in a previous lecture)

These DSLs are equipped with *tools that generate tools* to support the usage of the DSL under development

- Example: editor generator for textual syntax models



Language Workbenches

Language workbenches are tools that support:

- Definition of language aspects
- Generation of tools for new languages (e.g. editors)

A language workbench may support only a subset of the relevant aspects

Some existing workbenches provide General Purpose Language (GPL) for defining some language aspects

- Example: *Xtext* is a workbench that uses Java for validation, scoping, generation and other tasks

Everything is a Model (Again)

In the model-based approach for defining DSLs, the DSLs for the language aspects follow the same principles of definition:

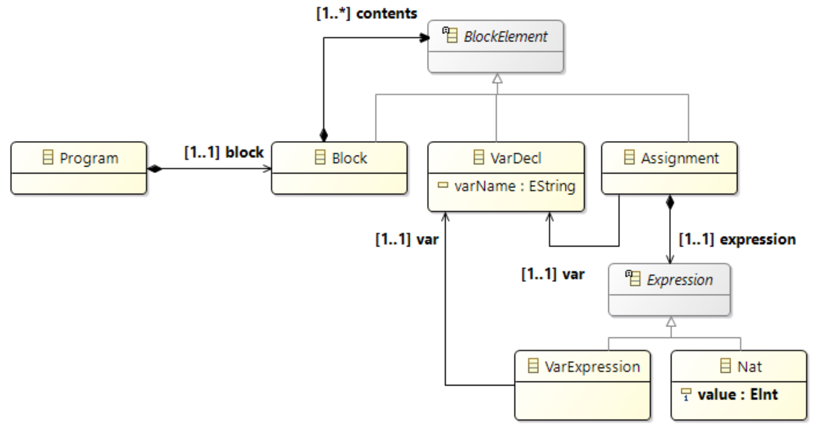
- They have a metamodel
- They have relevant aspects, set of tools etc.

Xtext

- Bridging Textual Concrete and Abstract Syntax in Modelware
- Xtext overview
- Xtext grammar language
- Modularity at the level of models
- Validation
- Scoping

Textual Concrete Syntax

Consider a simple metamodel:



Models can be created, for example:

- in XMI
- as Java objects

Much more natural and usable way is to just write text:

```
{ var a
  a = 2
  { var a
    a = 3
  }
}
```

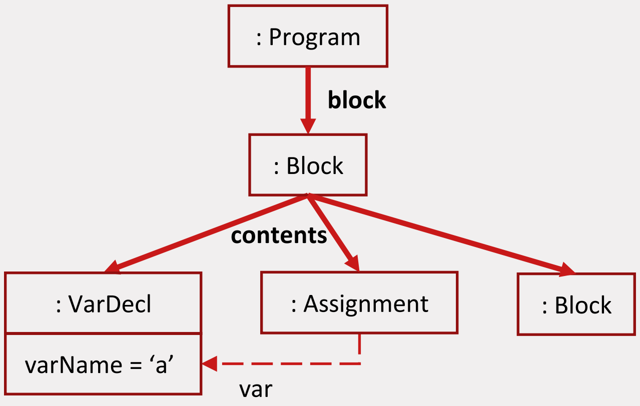
Textual Concrete Syntax

What we need is a *textual concrete syntax* for the small language

The way to define it is via a *grammar*:

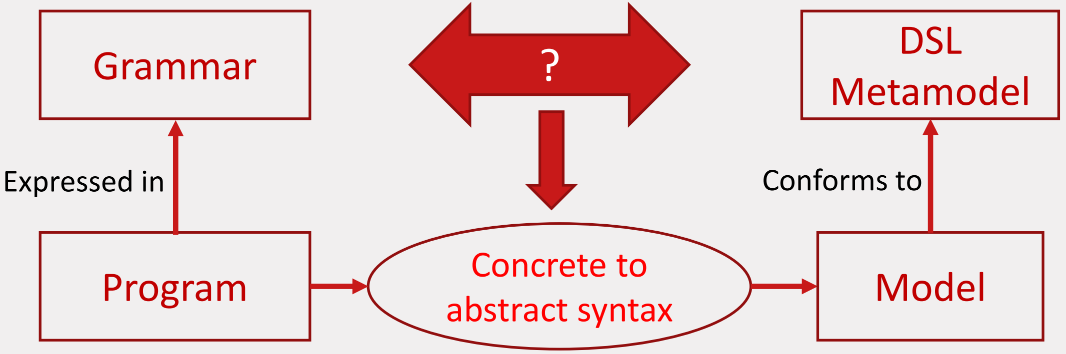
```
Program ::= Block

Block ::= ‘{’ (VarDecl |
               Assignment |
               Block)+ ‘}’

VarDecl ::= ‘var’ ID

Assignment ::= ID ‘=’ Expr

Expr ::= Nat | VarExpr

VarExpr ::= ID
```

Still, when we are processing the models, we want to work against the abstract syntax (the metamodel)



`{ var a a = 2 … {…} }`

From Concrete to Abstract Syntax

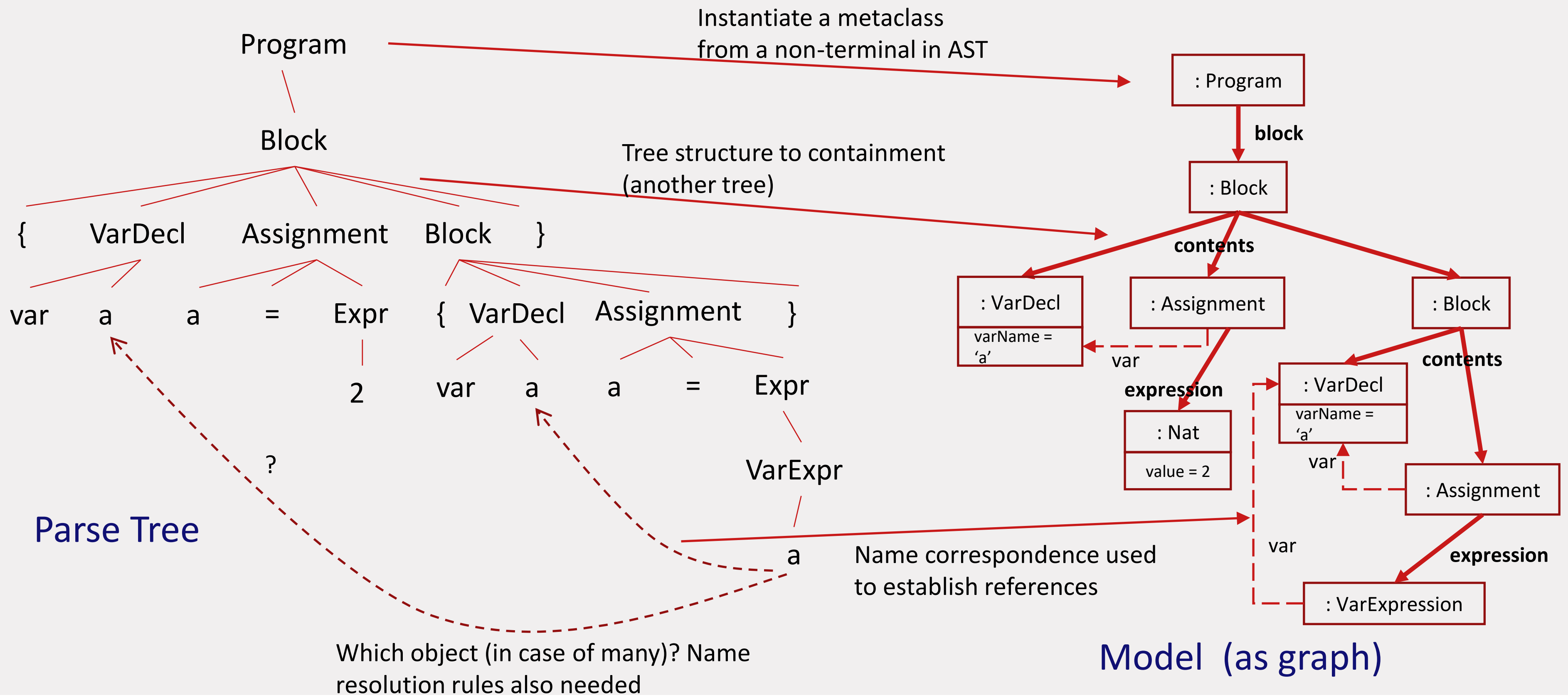


Such a translation builds upon the existing knowledge in Grammarware (e.g. parser generators)

- What are the main ingredients needed to automatically generate the translator?

A major requirement in the Modelware approach to DSLs is to *automate* the development of the *translator* from *concrete* to *abstract* syntax as much as possible

From Concrete to Abstract Syntax



Xtext Overview

Xtext allows definition of the textual concrete syntax of a DSL

- Provides a DSL for CF grammars
- For a given Ecore metamodel, grammars can be annotated with metaclasses and structural feature names
- Default and customizable name resolution rules

Automatically generates:

- *Editor* (with auto-completion, highlighting, etc.) and a *parser*
- Ecore model instance from a model in textual form (parsing, CS to AS conversion)

Xtext Overview

Built on top of ANTLR (parser generator technology, www.antlr.org)

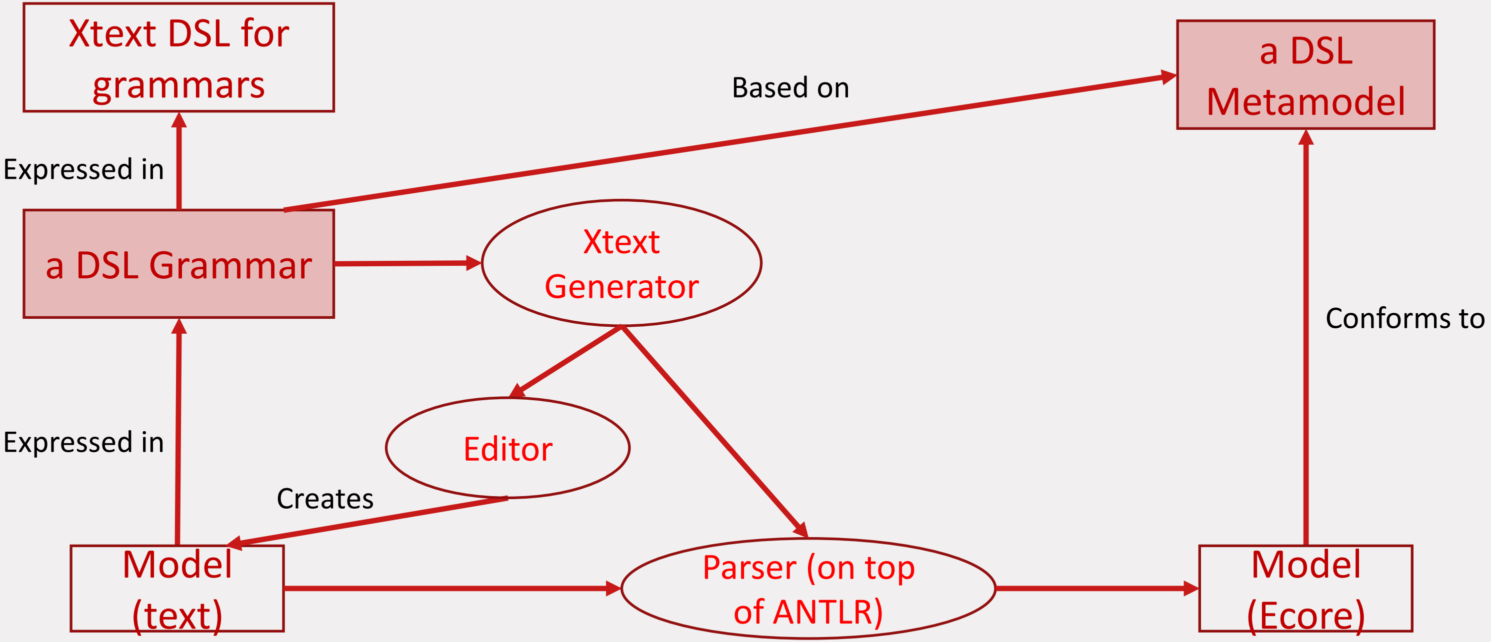
- An ANTLR grammar is generated from the defined Xtext grammar
- Parser is generated from the ANTLR grammar
- Once parsed, the programs are translated to Ecore models

Restrictions on the grammars (due to the underlying restrictions of ANTLR)

- No ambiguity
- No left recursion
- Only left factorized grammars

Xtext Overview

Xtext tool chain: main input is a *DSL grammar* based on the *DSL metamodel*



Xtext as a Language Workbench

Xtext is also a complex framework for developing textual DSLs

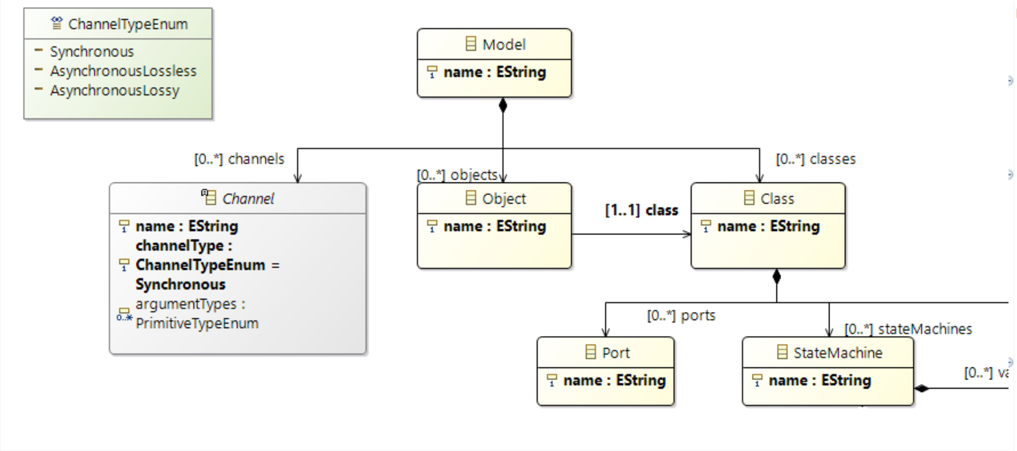
- Other language aspects can be defined, hence it is perceived as a language workbench

Other supported language aspects:

- Validation: validation framework based on Java
- Scoping: user-defined scoping rules
- Code generation: in language called Xtend (compiled to Java)

Xtext Grammar Language

The grammar language will be explained using the SLCO grammar as an example




```
model SimpleCalculator {
    classes

    Calculator {
        ports in out

        state machines
        Main {
            variables
            Integer a
            Integer b

            initial S

            transitions
            read_calculate from S to S {
                receive input(a, b) from in;
                send result(a + b) to out
            }
        }
    }
    objects
    c : Calculator
    u : User

    channels
    c0(Integer, Integer) sync from u.out to c.in
    c1(Integer) sync from c.out to u.in
```

```
Model returns Model:
      'model' name = ID '{' 'classes' classes += Class* 'objects' objects += Object* 'channels' channels += Channel* '}';

Class returns Class:
      name = ID '{' 'ports' ports += Port* 'state' 'machines' stateMachines += StateMachine* '}';
```

Xtext Grammar Language

```
Model returns Model:
	'model' name = ID '{' 'classes' classes += Class* 'objects' objects += Object* 'channels' channels += Channel* '}';
```

Extended Backus-Naur Form (EBNF) expressions:

- Rules are described using EBNF-like expressions
- There are four different possible cardinalities
    1. exactly one (the default, no operator)
    2. one or none (operator *?)*
    3. any (zero or more, operator **)*
    4. one or more (operator *+)*

The first rule defines the *start symbol* of the grammar

Xtext Grammar Language

Rule head

Metaclass to instantiate

```
Model returns Model:
        'model' name = ID '{' 'classes' classes += Class* 'objects' objects += Object* 'channels' channels += Channel* '}';
```

Attribute of the metaclass

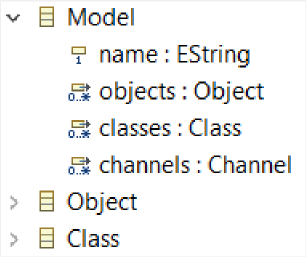
Containment reference with multiplicity > 1

A node in the parse tree derived from the rule will instantiate an object of the given *metaclass*

- Rule and metaclass names can be different

Values of *attributes* and *references* can be assigned

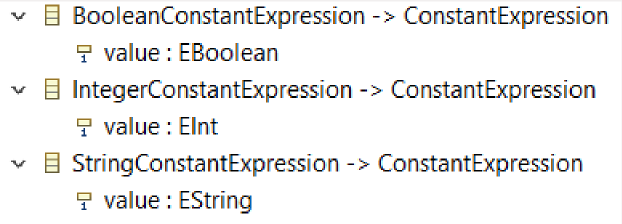
- assignment ‘=’: feature multiplicity 0 or 1
- assignment ‘+=’: the feature is a collection



Xtext Grammar Language

```
IntegerConstantExpression returns IntegerConstantExpression :
        value = INT;

StringConstantExpression returns StringConstantExpression :
        value = STRING;

BooleanConstantExpression returns BooleanConstantExpression :
        value = BOOL_LITERAL;
……

terminal BOOL_LITERAL returns ecore::EBoolean:

        'true' | 'false' | 'TRUE' | 'FALSE' | 'True' | 'False‘;
```

Predefined terminal rule for strings

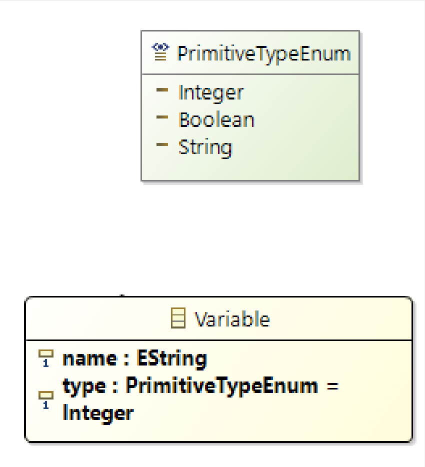


- Predefined terminal rules for STRING, INT, ID, single and multiline comments
  - Mapped to predefined Ecore types like EString and EInt
- Possibility to define custom terminal rules (as regular expressions in the general case):
  - Example: BOOL_LITERAL

Xtext Grammar Language

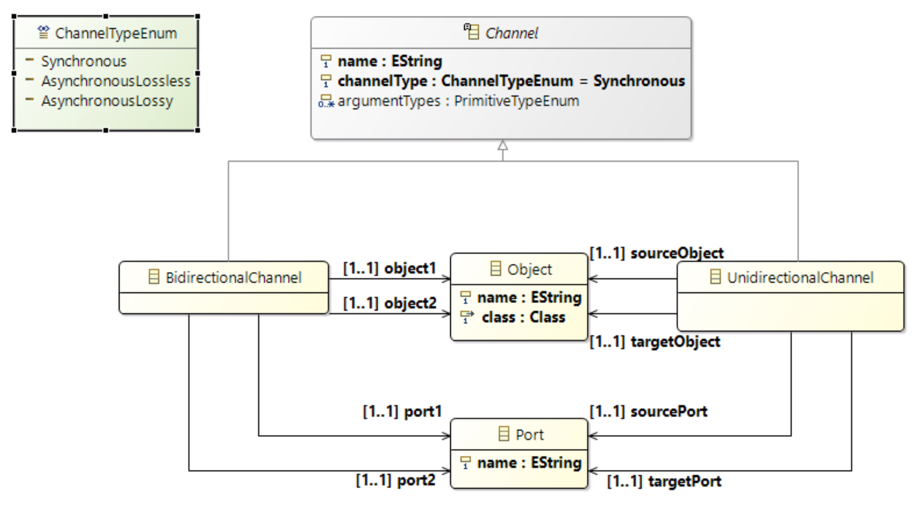
- Special kind of rules for enumerations

```
Variable returns Variable:
        type = PrimitiveType name = ID;

enum PrimitiveType returns PrimitiveTypeEnum:
        String = 'String' |
        Integer = 'Integer' |
        Boolean = 'Boolean‘ ;


terminal ID:
'^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```



Xtext Grammar Language

- Handling generalization relation
- Grammar fragments



Generalization represented as choice

```
Channel returns Channel:
        BiChannel | UniChannel;

BiChannel returns BidirectionalChannel:
        ChannelFragment 'between' object1 = [Object | ID] '.' port1 = [Port | ID] … ;

UniChannel returns UnidirectionalChannel:
        ChannelFragment 'from' sourceObject = [Object | ID] '.' sourcePort = [Port | ID] … ;


fragment ChannelFragment returns Channel:
        name = ID ('(' argumentTypes += PrimitiveType (',' argumentTypes += PrimitiveType)* ')')?
        channelType = ChannelTypeEnum;
```

Fragments are reusable grammar snippets. Do not lead to instantiation

Xtext Grammar Language

```
Object returns Object:
        name = ID ':' class = [Class | ID];

UniChannel returns UnidirectionalChannel:
        ChannelFragment 'from' sourceObject = [Object | ID] '.‘
        sourcePort = [Port | ID]
        'to' targetObject = [Object | ID] '.'
        targetPort = [Port | ID] ;
```

Cross-reference

```
objects
c : Calculator
u : User

channels
c0(Integer, Integer) sync from u.out to c.in
c1(Integer) sync from c.out to u.in
```

Cross-reference:

- Results in assigning values to non-containment Ecore references in the model
- In the program text the value appears as string but after parsing, a value for the reference is assigned taking an object in the AST that have a *name* feature with the same value (default mechanism)

In fact, Xtext combines context-free parsing and a form of semantic evaluation (name resolution)

Xtext Grammar Language

Unordered groups:

- The elements of an unordered group can occur in any order but each element can occur at most once
- Unordered groups are separated with '&'

Assignment to a Boolean feature. ‘true’ is assigned if the token is present

```
Modifier: static ?= 'static' &
           final ?= 'final' &
          visibility = Visibility;

enum Visibility: PUBLIC='public' | PRIVATE='private‘ | PROTECTED='protected';
```

allows:

```
public static final
static protected
final private static
public
```

Adding Modularity to Models

Let's extend SLCO:

- Add support for importing SLCO classes from a given model into another model

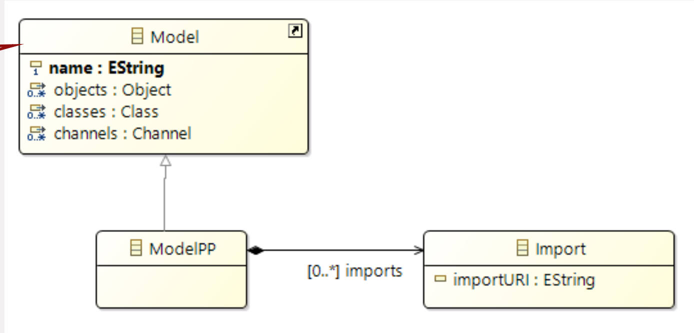
SLCOPP (SLCO plus plus) will extend *SLCO* and will add this new feature

- In effect, we allow *modularization* of user models
- Furthermore, we will reuse most of the original SLCO Xtext grammar, thus illustrating *grammar inheritance in Xtext*

Adding Modularity to Models

```
ModelPP returns ModelPP:
        'model' name = ID '{'
        imports += Import*

        'classes'
        … same as for SLCO Model … ;

Import:
        'import' importURI = STRING;
```

From SLCO metamodel



SLCOPP defines only two classes

The attribute *importURI* is interpreted by Xtext in a special way:

- The values point to other models
- The content of these models will be imported automatically
    - Can become values of references in the importing model

Adding Modularity to Models

```
model A {

   classes

   Calculator {
      ports in out
      …
   }

   objects

   channels

}
```

File *example1.slcopp*

```
model B {
   import ‘example1.slcopp’

   classes

   User {
      ports in out
      …
   }

   objects

   c : A.Calculator
   u : User

   channels

}
```

By default, Xtext uses a *fully qualified name* to identify the imported objects. The name is formed by top-down traversal of the containment hierarchy and concatenating the values of attribute *name* (if present).

The fully qualified name of class *Calculator* thus becomes *A.Calculator*

Xtext Cross-referencing Revisited

```
model A {

   classes

   Calculator {
      ports in out
      …
   }

   objects

   channels

}
```

File *example1.slcopp*

```
model B {
   import ‘example1.slcopp’

   classes

   User {
      ports in out
      …
   }

   objects

   c : A.Calculator
   u : User

   channels

}
```

In SLCO Grammar:

```
Object returns Object:
       name = ID ':' class = [Class | ID];
```

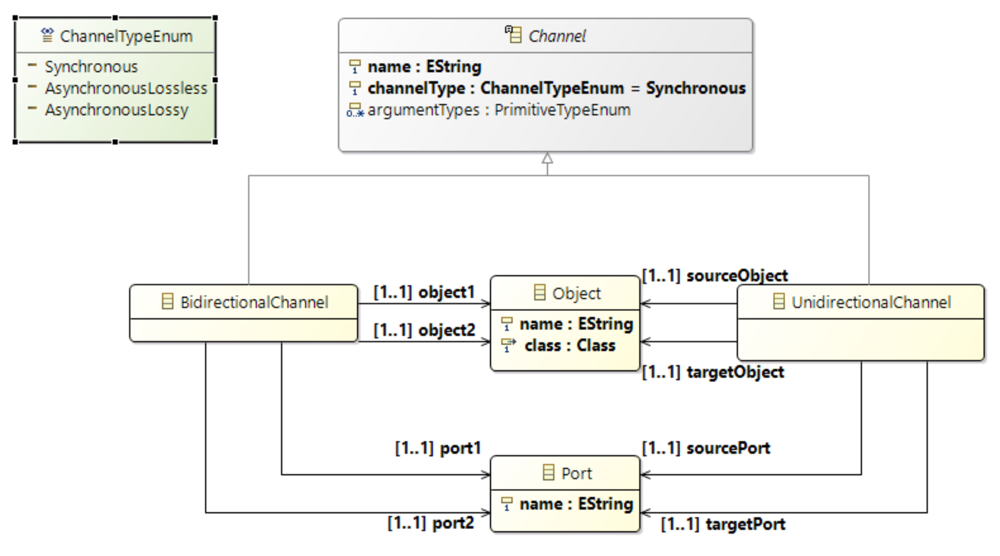
The type of the expected value

Optionally, the type of the token that represents the name to be used

In SLCOPP Grammar this needs to be defined differently:

```
Object returns Object:
       name = ID ':' class = [Class | FQN];

FQN: ID('.' ID)* ;
```

Customizing Name Resolution



How do we determine the value of references *sourcePort* and *targetPort* (port1 and port2 are similar)?

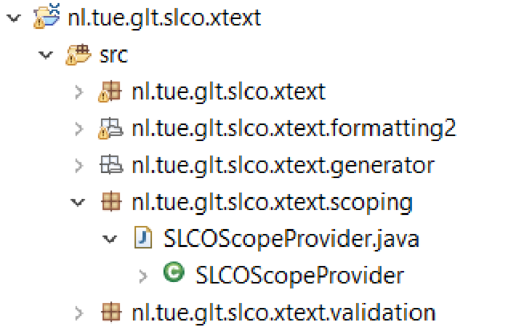
- *sourcePort* can only refer to a port that is defined in the class of the value of *sourceObject*
- The resolution therefore is applied not on all ports in the model but on a subset

Customizing Name Resolution

In Xtext, the name resolution can be customized by implementing (in Java) a new *scoping rule*

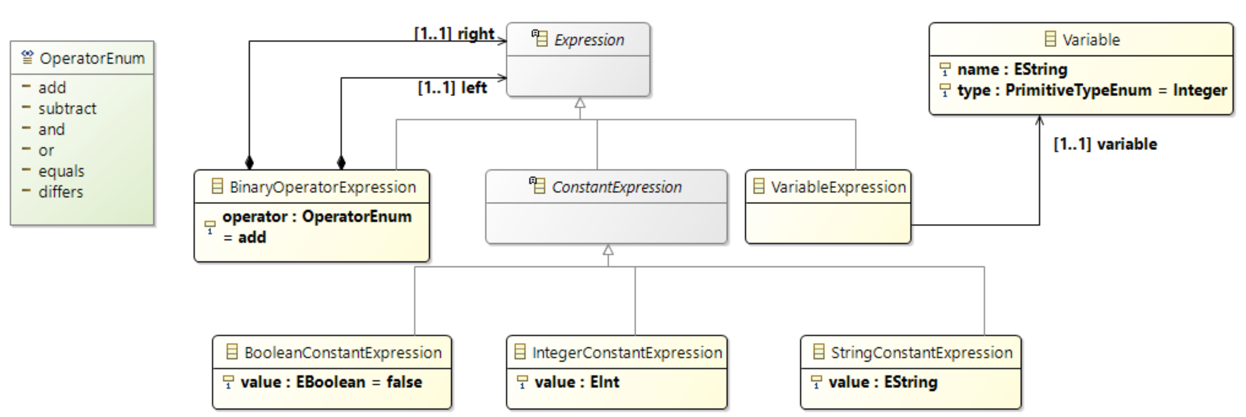
```
public class SLCOScopeProvider extends AbstractSLCOScopeProvider {

  public IScope getScope(EObject context, EReference reference) {

    if(reference == SlcoPackage.Literals.UNIDIRECTIONAL_CHANNEL__SOURCE_PORT) {
      return Scopes.scopeFor( ((UnidirectionalChannel)context).getSourceObject().getClass_().getPorts());
    }

    if(reference == SlcoPackage.Literals.UNIDIRECTIONAL_CHANNEL__TARGET_PORT) {
      return Scopes.scopeFor( ((UnidirectionalChannel)context).getTargetObject().getClass_().getPorts());
    }

    return super.getScope(context, reference);
  }
```

Creates a set of candidate target ports from the passed collection



Class *SLCOScopeProvider* is automatically generated. Method *getScope* needs to be implemented to supply scoping rules

Defining the Syntax of SLCO Expressions



A straightforward grammar based on the metamodel could be (not an Xtext syntax!):

```
Expr ::= ‘true’ | ‘false’ | INT | STRING | ID | Expr ‘add’ Expr | Expr ‘and’ Expr | …
```

This grammar is *left-recursive* and not supported by Xtext

Defining the Syntax of SLCO Expressions

To avoid the recursion, layering has to be applied. The following are the rules in Xtext syntax:

Concrete class to be instantiated

Handles:
Exp or Exp or …

Handles:
Exp and Exp and …

```
Expression returns Expression :
        ExpressionLevel1;

ExpressionLevel1 returns Expression:
        ExpressionLevel2 ({BinaryOperatorExpression.left = current}
                            operator = OperatorOrEnum right = ExpressionLevel2)* ;

ExpressionLevel2 returns Expression:
        ExpressionLevel3 ({BinaryOperatorExpression.left = current}
                            operator = OperatorAndEnum right = ExpressionLevel3)* ;


ExpressionLevel3 returns Expression:
        … continue with comparators

ExpressionLevel4 returns Expression:
        … continue with arithmetics
```

Handles:
Exp == Exp

Handles:
Exp + Exp

Defining the Syntax of SLCO Expressions

Finally, the lowest level is for the atomic expressions (constants and variable)

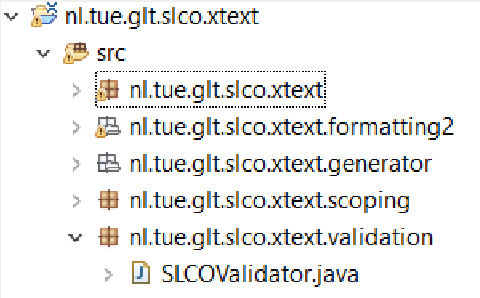
```
ExpressionLevel5 returns Expression :
        IntegerConstantExpression |
        BooleanConstantExpression |
        StringConstantExpression |
        VariableExpression ;
```

The layering follows the priority of evaluation

The example also illustrates that multiple grammar rules can produce objects of the same type; rule names are independent from the names of metaclasses

Model Validation with Xtext

OCL can be used to define validity rules; the validation can be invoked from the generated Xtext editor



It is also possible to implement the validation rules in Java:

- Xtext generates validator class, in our example *SLCOValidator*
- Users can write validation methods in the context of a metaclass

Model Validation with Xtext

Annotation indicates this method will be used for validation check …

… in the context of *Model* instances

```
@Check
public void checkDuplicateClassNames(Model model) {
  Set<String> names = new HashSet<String>();
  Set<Class> duplicates = new HashSet<Class>();

  for(Class c : model.getClasses()) {
    if(! names.add(c.getName())) {duplicates.add(c);}
  }
  for(Class c : duplicates) {
    error("Duplicate class", c, SlcoPackage.Literals.CLASS__NAME);
  }
}
```

The object that has an error

Inherited method. Will show an error message in the editor

Ecore feature that will be highlighted

Apart from errors, there are methods for warnings

Other Xtext Features

Xtext can *derive* (infer) a metamodel from a grammar:

- The same grammar language is used
- Some support for controlling the inference process

Not every metamodel can be easily ‘fit’ to the desired grammar

- Xtext works well if there is a large degree of structural similarity between the grammar and the metamodel
- Possible workaround: derive a metamodel from the desired grammar (Concrete Syntax metamodel), transform the models to ones that conform to the desired metamodel

Demo

- The Xtext grammar of SLCO and SLCOPP
- The generated editor
- Scoping and validation

Summary on Xtext

- Good industrial support, well integrated with Eclipse and EMF
- Many non-trivial real-life languages has been implemented with Xtext
- Powerful framework: users can customize almost every aspect and default support (e.g. importing, scoping, editor features)
  - But also complex, documentation on advanced features is scattered and often not updated

Summary on Xtext

In general, some important features are lagging behind the available theoretical knowledge and research prototypes

- Very limited form of language extension (single grammar inheritance)
- Restricted parsing: LL class, based on ANTLR; more powerful approaches exist, e.g. GLL, GLR (exemplified by Rascal)
- Modularity, if needed, has to be defined for every new DSL (although some basic support is present)
- Important language aspects such as typing, validation, scoping are implemented in a GPL (Java) although more compact specialized notations exist in the literature

Resources

Xtext home page

- https://www.eclipse.org/Xtext/

“Implementing Domain Specific Languages with Xtext and Xtend – Second Edition”, Lorenzo Bettini, 2016 (not freely available)