Open Source Modeling

Textual DSLs with oAW Xtext

Bernd Kolb
b.kolb@kolbware.de
http://www.kolbware.de

Markus Voelter
voelter@acm.org
www.voelter.de
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

© Bernd Kolb, Markus Voelter 2008
Intro to Textual DSLs
Getting Started with Xtext
Intro to Example
Defining the Grammar
Simple Editor Customization
More Advanced Customization
Processing the Models
Summary
A Domain Specific Language (DSL) is a **focused, processable language** for describing a specific **concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

**DSLs can be classified** in many ways:
- Configuration vs. Customization
- Internal vs. External
- Graphical vs. textual

**We concentrate on external textual customization** DSLs
Typically textual DSLs are build using one of the many *parser generators* (ANTLR, JavaCC, Lex/yacc), or even by a hand-writing a custom parser.

Parser: *match text* and try to create a *parse tree*

Parser Generator: Generate a parser from a *grammar*
Typically, transformed into an Abstract Syntax Tree (AST)
- No whitespace, reified nodes, often binary tree

The AST can be considered the **model**
- The node types of the AST act as the **metamodel**

Programs processing the “sentence” are typically written against the AST – they usually don’t care about the parse tree
Challenges

- Writing a parser is **non-trivial**
- Using a **parser generator** makes it easier, but still not for everybody
- Also: out of the box, a parser generator **only creates a matcher** and/or a **simplistic AST**. You still need to
  - Transform the model into a form that is easily processable
  - Create an editor with syntax highlighting and code completion

→ A lot of work! Only few people are willing to do that
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary
Xtext

- Xtext makes this much more feasible.
- Based on an **EBNF grammar** it generates
  - ANTLR3-based **parser**
  - EMF-based **metamodel**
  - Eclipse **editor** with
    - syntax highlighting
    - code completion
    - customizable outline
    - code folding
    - real-time constraint checking
- Xtext is part of **openArchitectureWare / Eclipse Modeling**
Creating a new Xtext Project I: Xtext Project Wizard

- **Main Project Name:**
  - name of language project
  - `.editor` project contains editor

- **Language Name:** package
  - name of generated meta model

- **DSL File Extension:** file extension used to associate editor

- **nsURI:** full namespace URI of the generated meta model

- **Base Package:** all code will be in or below this package

- **Code Gen Project:** See last section of this presentation
Creating a new Xtext Project II: Created Language Project

- **Language Project**
  - Contains a **parser** to be used in standalone programs
  - Put your language’s **constraint checks** here
  - This contains your metamodel’s **extension** functions
  - Define custom **linking** functionality here
  - Run this to **generate** metamodel, editor, and everything else
  - Contains a number of **properties** (e.g., those you put into the wizard)
  - Specify your **grammar** rules here
  - Generated „base files“ - don’t change.
  - Generated **metamodel**
  - Generated **Xtext Rules** file

© Bernd Kolb, Markus Voelter 2008
Creating a new Xtext Project III: Created Editor Project

- Editor Project
- Customize Code Completion here
- Customize outline labels and icons here
- Define go to definition and find references here
- Provide customized outline structure and viewpoints here
- Customize keyword font style here
- Generated „base files“ - don’t change.
- Put customized icons (for the outline view) here

© Bernd Kolb, Markus Voelter 2008
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary
DSL for **importing data**: common problem in large enterprises (e.g. hospitals)

**Problem**: read data like the one on the right into Java Beans
- Example is about patients, stays, wards, insurance data

**Record Structure**:  
<Type> <Fd1>,<Fd2>, ... <Fdn>

**Different Record Types** have different fields
Intro to Example II: Solution Approach

- Define **data structures** (à la Java Beans)
  - Entities
  - Attributes, References (:1, :n)
- Define **parser**
  - Checks record order
  - Instantiates data structures
  - Maps fields to attributes
  - Assembles references
- Backend: **Generate** Java code for **beans** as well as **parser**

© Bernd Kolb, Markus Voelter 2008
Intro to Example III: Solution Tooling
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

© Bernd Kolb, Markus Voelter 2008
Grammar is defined in an EBNF-like format in the Xtext editor.

The editor provides code completion and constraint checking for the grammars themselves.

Grammar is a collection of Rules. Rules start with their name followed by : and ending with ;
Different Kinds of Rules

- **Type Rule**
  - For each rule Xtext creates a class in the metamodel
  - Each rule property results in a property of the metaclass

- **String Rule**
  - String rules are parsed to a String.
  - They are in effect custom lexer rules (recognizing string patterns)

- **Enum Rule**
  - Limited set of alternatives; mapped to an Ecore enum

- **Native Rule**
  - A lexer rule is directly interpreted by ANTLR; mapped to a String
Built-in Lexer Types (I)

- **ID**
  
  ![Diagram of ID type]

- **STRING**
  
  ![Diagram of STRING type]

- **INT**
  
  ![Diagram of INT type]
- Multiline comment

```
/*
\u0000...\uE000
*/
```

- Single line comment

```
//
//
\{\u0000...\u0008, \u0009...\u000D, \u000E...\uE000}\n``` 

- Whitespace

```
\{\t...\n, \r, \f}\n``` 

- The content of those rules is **not transformed** into the meta model
- **Reference**

  - `[`  ID  `]`
  - `|`
  - `ruleName`
  - `]`

- **File Reference/Import**: Xtext derives the default Linking information from here (more later)

  - `'URI'`
Type Rules

- A **type rule** is mapped to a metaclass in the metamodel.
- It may contain **keywords** (using string literal syntax).
- It also contains **properties** which will result in properties of the respective metaclass.
  - The property type is derived from the called rule.

- There are different **kinds of properties**
  - = (single assign)
  - += (multiple assign/add)
  - ?= (boolean assign)

- There are different **property cardinalities**
  - ? (0..1)
  - * (0..n)
  - + (1..n)
  - nothing (1..1)
Type Rules – Combinations

property name ->

= ->

Type Rule ->

String Rule ->

Build-In Rule ->

Native Rule ->

nothing (1..1)

+= ->

Type Rule ->

String Rule ->

Build-In Rule ->

Native Rule ->

? (0..1)

?= ->

Keyword ->

nothing (1..1)

->

* (0..n)

+ (1..n)

nothing (1..1)

© Bernd Kolb, Markus Voelter 2008
Type Rules – Examples

- **ImportSpec:**
  - **Keywords:** `import`, `ext`, `=`, `{ and }`
  - **Expects** one occurrence of `DataSection` rule assigned to the `dataSection` property

- **DataStructure:**
  - **Expects** any number of `Attribute` or `Reference` occurrences and assigns them to the `attributes` and `references` property, respectively
- By default, execution of rules results in a tree of data
- You can reference other elements via a reference rule
  - In textual languages a reference has to be by name
  - During linking, Xtext “dereferences” these by name-references
  - Target’s name property is used here (customization: see later)

- Example: reference to a DataStructure. ID is is the lexer used for the referencing token
A resource reference can be used to import another model file (typically of the same DSL)

When customizing e.g. content assist, Xtext automatically provides a global view on the contents of all imported files
Abstract Type Rules

- A type rule can be abstract
- An abstract type rule is basically a collection of OR-ed alternatives: R1 | R2 | R3
- Mapped to an abstract metaclass
  - The or-ed alternatives become concrete subclasses
  - Common properties of the alternatives are lifted into the abstract superclass
- The same rule can be or-ed in several abstract rules in which case the metaclass has several superclasses.
Property is the abstract rule.

Can be a Reference or an Attribute

The Data Structure has a collection of Properties, referencing the abstract rule

In the meta model, name has been be lifted, type has not
String Rules

- String rules are **lexer rules** (i.e. used to define rules for custom character sequences)
- Declared via the *String* keyword
  
  ```
  String qualifiedName:
  ID ("." ID)*;
  ```

- String rules can be **composed** from other lexer rules.
  - Other String rules
  - Built-in lexer rules (e.g. ID)
  - Native rules (see Xtext Reference Manual)
Enum Rules

- A Enum Rule is used to define a **limited set of defined alternatives**
- It is **mapped to an Enum** in the metamodel
- It is declared via the *Enum* keyword and contains **Enum Literals**
  - An Enum Literal has a token name and a string representation

```plaintext
Enum RecordType:
    A001="a001" | A002="a002" | P001="p001";
```

- It can be used just like any other rule.
A native rule contains a string which is passed to ANTLR without further processing it.

It is typically used to define lexer rules that cannot be expressed using Xtext syntax
- E.g. whitespace-aware lexer rules, such as define custom comment syntax

It is declared via the Native keyword

The example introduces a comment (# till end of line) that is preserved into the model
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- **Simple Editor Customization**
- More Advanced Customization
- Processing the Models
- Summary

© Bernd Kolb, Markus Voelter 2008
Sidebar: Xtend

- **OCL-like expression language** used throughout oAW
  - Can be used in constraint checks, model transformations and generators
  - **Add “methods”** to meta types (Java calls are possible if necessary)
  - **Path expressions**, **set operations**, (some) **higher order** functions
  - **Polymorphism** (multiple dispatch)
  - **Tool support**
    (syntax highlighting, code completion, debugger)
- Uses the **oAW Check language**, based on Xtend
  - Used to validate the static semantics of the models
- Report Warnings and Errors
- Based completely on the **generated meta model**, no concrete syntax hassles
- Tool support (syntax highlighting, code completion, ...)

```plaintext
import imp;
extension dataimport::dsl::Extensions;

context Instance ERROR "type not defined":
  allElements().typeSelect(DataStructure).contains(type);

context FieldMapping ERROR "instance not defined or not in scope":
  ((RecordHandler)cContains()).instancesInScope().contains(instance);

context FieldMapping ERROR "field not defined or type ":instance.type.name:
  instance.type.attributes.contains(field);

context Instance ERROR "instanc names must begin with a lowercase letter":
  name.toFirstLower() == name;
```
The constraints defined for the language are evaluated in the editor on the fly.

Constraints can also be checked as part of a workflow that processes the models.
Xtext supports the customization of the **Outline View** structure and icons/labels.

Labels and icons are changed by **overriding** `label(...)` and `image(...)` for your meta types.

Actual image files (.gif) go into editor project’s **icons** directory.

© Bernd Kolb, Markus Voelter 2008
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- **More Advanced Customization**
- Processing the Models
- Summary
By default, the **structure of the outline** view corresponds to the structure of the document.
- This default structure can be **changed**
- **Additional** structures ("viewpoints") can be defined

- **Default Viewpoint**

- **Types Viewpoint**
  (shows data types and instances)
Customizing the Outline II: Structural Customization II

- Use naming convention to define a `outlineTree` function for a given viewpoint.
- Use `create` extensions to construct the tree and create `UIContentNodes`.
- Use these functions to parameterize the created `UIContentNodes`.
- Connects to the original file for selection highlight.
- Return list of viewpoint IDs.

```java
import imp;
import tree;

extension dataimport::dsl::EditorExtensions;
extension dataimport::dsl::GenOutline reexport;

create UIContentNode outlineTree_Types(emf::EObject model) {
    setLabel(model.label()) ->
    setImage(model.image()) ->
    setContext(model) ->
    children.addAll(model.eAllContents.typeSelect(DataStructure).create

create UIContentNode createDSContainer(DataStructure ds, emf::EObject model) {
    setLabel(ds.label()) ->
    setImage(ds.image()) ->
    setContext(ds) ->
    children.addAll(model.eAllContents.typeSelect(Instance).select(i1)

create UIContentNode createINode(Instance i):
    setLabel(i.label()) ->
    setImage(i.image()) ->
    setContext(i);

List<String> viewpoints() :
    "Types";
```

© Bernd Kolb, Markus Voelter 2008
Customizing Content Assist

- Content Assist / Code Completion is customized via overwriting certain extension functions:
  
  ```java
  List<Proposal> complete<MyType>_<propName>(emf::Eobject ctx, String prefix)
  ```

- A Proposal consists of a label, an icon and the text to insert

© Bernd Kolb, Markus Voelter 2008
By default, if you use references in your DSL, you can use **Go To Definition** and **Find References** out of the box (it works cross-resource, see next section)
To customize navigation, **overwrite the following extension** in the *Navigation.ext* file.
(you do NOT have to do this for DSL references!)

- **Find the Definition** of a “string” in a context:
  
  ```
  emf::EObject findDeclaration(String s, emf::EObject ctx)
  ```

- **Find the References** of a “string” in a context:
  ```
  List[UIContentNode] findReferences(String s, emf::EObject ctx)
  ```

- **Create the node** for the list of results:
  ```
  create UIContentNode createNode(RuleName tr)
  ```
It is possible to **customize the font style** for keywords (and only keywords)

You can **customize** bold, italic, underline, strikethrough as well as foreground and background colors

Override the **fontstyle() extension** for a specific meta class:

There is some **performance impact** when using this feature.
Tbd.

*Explain how to code-complete and constraint check into other EMF files (non-Xtext Stuff)*
Customizing the Linker

- The Linker is used to **wire different models**
- In most cases the **automatically created** grammar-derived Linker is sufficient
- However it is possible to **customize the behavior** further:
  
  Override `link(emf::EObject this)`
It is necessary in several cases to uniquely identify model elements by one of their properties. Used by

- Linker
- findReference/findDeclaration
- ...

Default property is name

To customize this, you can override the id() extension in the Extensions.ext file.
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

© Bernd Kolb, Markus Voelter 2008
Since Xtext instantiates Ecore metamodels, the models can be processed with any EMF tool – e.g. oAW.
  - We will not explain those here.

You can process them with oAW Workflows, use them like any other EMF resource and parse them into your own Java programs.

You can check the Create Generator Project checkbox in the projection creation wizard to get a sample generator as a starting point.
The only Xtext-specific aspect is using the generated parser.
Use EMF’s **native resource mechanism**

- Xtext **registers a ResourceFactory** for the DSL’s file extension
- Simply **pass the uri** of the file you want to read to `getResource()`
- Note that you **cannot write** the model back!
Use the generated parser

- This is what the `ResourceFactory` does internally with some default behavior for error handling

- Note how parse errors and constraint violations are handled.
Typically you are only working on the AST (ecore file).

However, sometimes you need data from the parse tree:
- Location of an element in the source file
- The text for a certain element
- The parse tree node at a certain offset

Here is an example of how to get the location of an element in the source file:
Contents

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary
Designing good languages is not easy – and Xtext cannot simplify that process.

However, building good tools for good languages is also not easy, and often a reason for not trying to build a language.

Xtext makes that second challenge – the tooling – much much easier, thereby making language development a real practical option.

Go ahead, give it a try!
oAW Xtext is currently in the process of transitioning into the (relatively) new **Textual Modeling Framework** project.

Together with INRIA’s TCS, it **serves as the basis** for TMF.

See also
Limitations

- It is not yet possible to specify **custom action code**
  - This sometimes results in “ugly” meta models
  - Becomes apparent when building expression languages
  - [This is a plan item for TMF]

- It is not yet possible to **start from a metamodel** and annotate the notation
  - The current approach is suitable if you only use a textual syntax
  - If you want to mix syntaxes or want to define a textual syntax for an existing metamodel, the current approach is not useful
  - [This is a plan item for TMF]
More...

- [www.openarchitectureware.org](http://www.openarchitectureware.org)
  - Specifically the reference docs contain more info about Xtext

- [www.eclipse.org/modeling/](http://www.eclipse.org/modeling/)
  - EMF and all the other related tooling
  - Future home of Eclipse TMF

- Podcast: Software Engineering Radio
  - [www.se-radio.net](http://www.se-radio.net)
  - Several Episodes on MDSD
Thanks for your attention!

Questions?

Bernd Kolb
b.kolb@kolbware.de
http://www.kolbware.de

Markus Voelter
voelter@acm.org
http://www.voelter.de

© Bernd Kolb, Markus Voelter 2008