Voces

Ştefan Apostoaie, Alexandru Chica, Marina Ghiucă, Irina Roznovăţ

Computer Science Faculty, Iaşi

**Abstract.**

This paper is documentation for a project that creates an extension for the Eclipse IDE which allows editing of XML/XHTML documents with embedded metadata vocabularies. The editor supports XML/XHTML syntax and is able to dynamically import vocabularies like FOAF, DOAP. RDF parsing of vocabularies is done with Jena. The editor consumes web services which expose vocabulary syntax. All information about this project can be taken from its site: <https://code.google.com/p/voces/>.

**Keywords:** meta data, vocabulary, Xtext, eclipse, DSL

Introduction

Editor

Xtext

Xtext is a framework for development of domain specific languages (mainly programming languages). Xtext allows the user to define a language grammar using EBNF syntax. Starting from this grammar, the framework generates a parser and an Abstract Syntax Tree meta-model, together with a fully featured default editor for Eclipse. Xtext integrates with Eclipse Modeling technologies such as EMF, GMF, M2T, and parts of EMFT, so that adding new features for a DSL becomes an easy task.

Editor grammar and meta-model

For the Voces project, the following grammar was created:

**grammar** ro.fii.wade.voces.Metavoc **with** org.eclipse.xtext.common.Terminals

**generate** metavoc "http://www.fii.ro/wade/voces/Metavoc"

XMLModel :

(contents+=XMLValidElement)\*

;

XMLValidElement **returns** *XMLValidElement* :

(startelement=XMLStartElement)

( (children += XMLValidElement)\* | name=ID | content=INT )

(endelement=XMLEndElement)

;

XMLStartElement **returns** *XMLStartElement* :

"<" XMLElementText ">" ;

XMLEndElement **returns** *XMLEndElement* :

"</" XMLElementText ">" ;

XMLElementText **returns** *XMLElementText* :

namespace=VocNS

":"

nselements=NSElement

(attributes=XMLElementAttributes)\*

;

XMLElementAttributes **returns** *XMLElementAttributes* :

namespace=AttrNS

":"

nselements=NSElement

"="

elementValue='"'name=ID'"'

;

AttrNS **returns** *AttrNS* :

name=ID;

VocNS **returns** *VocNS* :

name=ID;

NSElement **returns** *NSElement* :

name=ID;

Basically this grammar defines the RDF/XML format. XMLModel is the root node of the syntax tree, which can contain zero or more XMLValidElement rules.

An XMLValidElement rule can contain a start element rule (<element>), other XMLValidElement rules or plain text (name=ID) and an end element rule (</element>).

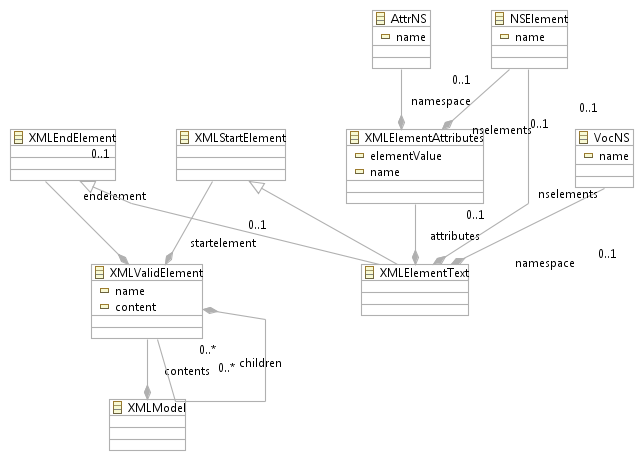
The rules XMLStartElement and XMLEndElement contain the required XML brackets (‘<’, ‘>’, ‘</’) and a rule called XMLElementText, which contain the syntax of the XML element text. A RDF/XML element may have a namespace name, followed by a ‘:’ character and then the namespace element; as an option, the element might have one or more XMLElementAttributes rules, which describe the syntax of an XML attribute (namespace:element=”value”).

In the grammar specification, you can see the line “name=ID” referenced many times. This specifies that the rule accepts any character the user types in. the ID terminal is defined as:

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

These are default terminals provided in org.eclipse.xtext.common.Terminals grammar.

From the grammar specification above, the meta-model in Fig.1 is generated.



**Fig. .** Meta-model generated from grammar

The meta-model is used to generate the other classes needed in the application: default content assist, highlighting and verification classes.

Content highlighting extension

To customize the content highlighting in the editor, the package ro.fii.wade.voces.highlighting was added in ro.fii.wade.voces.metavoc.ui/src. This package contains the two classes that are needed to implement semantic highlighting for the model: MetavocSemanticHighlightingConfiguration and MetavocSemanticHighlightingCalculator. The first class specifies the highlighting configuration: the style and appearance of the elements; this class has to implement the ISemanticHighlightingConfiguration interface.

To add a specific highlighting configuration, the user has to override the configure method from the interface. It’s implementation is as follows:

@Override

**public** **void** configure(IHighlightingConfigurationAcceptor acceptor) {

// **TODO** Auto-generated method stub

acceptor.acceptDefaultHighlighting(*NAMESPACE\_ID*, "Namespace", namespaceType());

acceptor.acceptDefaultHighlighting(*NAMESPACE\_ELEMENT\_ID*, "NamespaceElement", namespaceElementType());

acceptor.acceptDefaultHighlighting(*NAMESPACE\_SEPARATOR\_ID*, "NamespaceSeparator", namespaceSeparatorType());

}

In order to register the components within the IDE, in the class MetavocUIModule, the following methods have to be added:

//bind the calculator in the existing UI

**public** Class<? **extends** org.eclipse.xtext.ui.common.editor.syntaxcoloring.ISemanticHighlightingCalculator> bindSemanticHighlightingCalculator() {

**return** MetavocSemanticHighlightingCalculator.**class**;

}

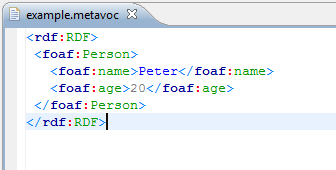
//bind the configuration in the existing UI

**public** Class<? **extends** org.eclipse.xtext.ui.common.editor.syntaxcoloring.ISemanticHighlightingConfiguration> bindSemanticConfiguration() {

**return** MetavocSemanticHighlightingConfiguration.**class**;

}

The result is showed in Fig.2.



**Fig. .** Semantic highlighting

Content assist extension

UI Plug-in

Web service

Voces Parser

In **Voces**, vocabularies are parsed using Jena. Jena is a Java API for semantic web applications. It was developed by Brian McBride into Hewlett-Packard laboratories and it is derived from SiRPAC. It can be used to create and manipulate RDF models. Jena provides some interfaces for RDF triplets’ manipulation.

The Jena "Statement" interface provides methods for getting and/or setting the subject, predicate and object of a statement. The object of a statement can be either a resource or a literal and the getObject() method returns an object typed as RDFNode, which is a common superclass of both Resource and Literal. An object type can be determine using instanceOf() method.

RDFNode interface provides a common base for all the elements that can be part from a RDF triple. Literal interface refers to literals and strings that are <object> from the RDF triple. The objects that implements Container, Alt, Bag or Seq interfaces can be also seen as <object> into RDF triples. Objects that implements Property interface can be RDF triples’ predicates.

To create a simple RDF/XML document, Model interface have to be used. ModelMem class creates a RDF model into memory. This class extends ModelCom class that contains all the methods for models’ usage. The ModelRDB class is used to manipulate those RDF models that are stored into relational databases as MySQL, Oracle or PostgreSQL. Against ModelMem models, the ModelsRDB models are persistent.

RDF models are directly accessed through iterators: NodeIterator (for generic nodes – also resources and literals), ResIterator and StmtIterator.

Another way to access information from a RDF model is using SPARQL – query language for RDF. Results are RDF triples or sets of RDF graphs.

Voces Parser uses Jena to parse vocabularies that are received through its WEB service. Only “namespace” and “subjects” properties are extracted from vocabularies and stored into xml files. These output xml files are provided by Voces Parser to web service and later processed by it.

RDF packages used by Voces Parser are:

**import** com.hp.hpl.jena.rdf.model.Model;

**import** com.hp.hpl.jena.rdf.model.ModelFactory;

**import** com.hp.hpl.jena.rdf.model.ResIterator;

**import** com.hp.hpl.jena.rdf.model.Resource;

The following code shows exactly how the RDF model is created with the information from the vocabularies, how its content is processed and only “subjects” properties are extracted from the model.

// create an empty model

Model model = ModelFactory.createDefaultModel();

…

// read the RDF/XML file

model.read(in, "");

…

//take all subjects from a RDF-triplet

ResIterator iter = model.listSubjects();

//process all the subjects

while (iter.hasNext())

{

Resource subject = iter.nextResource();

String resourceInfo = subject.toString();

…

}

For output file name, parser uses a template: File + id + xml extension. For example, being given as input the **FOAF** vocabulary (*http://www.foaf-project.org/*), the output will be File1.xml with the following values:

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<Vocabulary>

  <Namespace>http://xmlns.com/foaf/0.1</Namespace>

  <Terminal>knows</Terminal>

  <Terminal>firstName</Terminal>

  <Terminal>icqChatID</Terminal>

  <Terminal>birthday</Terminal>

  <Terminal>givenname</Terminal>

…

</Vocabulary>

Editor

Conclusion

References

[1] <http://www.eclipsepluginsite.com/>

[2] <http://www.vogella.de/articles/EclipsePlugIn/article.html>

[3] <http://agile.csc.ncsu.edu/SEMaterials/tutorials/plugin_dev/>

[4] <http://jena.sourceforge.net/tutorial/RDF_API/index.html>

[5] <http://jena.sourceforge.net/IO/iohowto.html>

[6] <http://www.devx.com/semantic/Article/35906>

[7] <http://willdaniels.co.uk/reference/rdf-vocabulary>

[8] <http://www.foaf-project.org/>

[9] <https://trac.usefulinc.com/doap>

[10] <http://dublincore.org/specifications/>

[11] <http://www.java-forums.org/swt/9751-swt-list-example-demonstration.html>