Variability in Xtext DSLs

Markus Voelter, voelter@acm.org

Version 0.7

This documentation explains how to use the tooling provided as part of the *dslvariantmanagement* Google Code project. It supports the expression of negative variability in arbitrary Xtext DSL.

This document really only contains the user guide, but no background information about the approach or concepts. Please read the following PDF instead:

http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf

http://www.voelter.de/data/articles/ArchitectureAsLanguage-Part2-PDF.pdf

Tooling

The variability tooling is currently implemented based on pure::variants (http://pure-systems.com).

You need to install pure::variants version greater 3.0.4, including the "pure::variants - Connector for Ecore" and "pure::variants - Connector for Source Code Management" feature. An evaluation version is available at http://www.pure-systems.com/pv-update.

Note that the Community Edition is not sufficient.

Feature Access

Create an Xtext Project

To get started, we create a new Xtext project. Just use all the defaults in the *New Xtext Project* wizard. Enter the following into the *Mydsl.Xtext* file as the language grammar.

```
grammar org.xtext.example.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/MyDsl"

System:
  (entities+=Entity)*;

Entity:
  "entity" name=ID "{"
          (attributes+=Attribute)*
  "}";
```

```
Attribute:
  name=ID ":" type=ID;
```

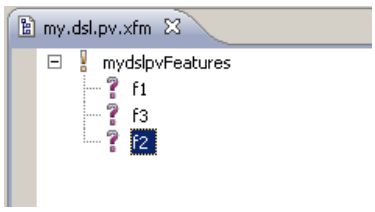
Generate the DSL, and verify that it works. Create a plugin project *org.xtext.example.mydsl.test* in the runtime workbench and put a *test.mydsl* file with the following content into the *src* folder.

```
entity E1 {
  a : String
}

entity E2 {

}
```

Create a p::v Project

The next step is to create a project that holds the p::v variant model and its EMF export. Create a new Variant Project (using the Wizard) named *org.xtext.example.mydsl.pv*. Then open the project properties and set the Feature Model Ecore Export to *PV Feature Model*.

In the variant model that has been created by the wizard in the project's root folder (*org.xtext.example.mydsl.pv.xfm*) create a couple of features. Make sure these are optional features!



Then save the model. A *org.xtext.example.mydsl.pv.xfm.xmi* next to *org.xtext.example.mydsl.pv.xfm* should appear. This contains the variant model in EMF form. Whenever you change something in the *.xfm* make sure you save it to trigger the export and update the *.xfm.xmi* file.

Access Features from DSL

Add the *org::openarchitectureware.var.featureaccess* plugin to dependencies of DSL project. Make sure you check the *reexport* flag. Then modify the grammar to include the red stuff.[1]

```
System:
  (featureModel=FeatureModelImport)?
  (entities+=Entity)*;

Entity:
  "entity" name=ID "{"
          (attributes+=Attribute)*
  "}";

Attribute:
```

[1] Because of the missing language modularization features of Xtext 4.x this could not be provided as a "library" in oAW 4.x. TMF Xtext provides better modularization support, but we didn't change the approach yet. Hence, copy and Paste is necessary.

```
  name=ID ":" type=ID (featureClause=FeatureClause)?;

FeatureClause:
  FeatureAndList | FeatureOrList | FeatureExpression | Feature ;

FeatureAndList:
  "featureAndList" (retained?="retain")? "(" featureList+=ID ","
featureList+=ID)* ")";

FeatureOrList:
  "featureOrList" (retained?="retain")? "(" featureList+=ID ","
featureList+=ID)* ")";

FeatureExpression:
  "featureExp" (retained?="retain")? "(" expression=OrExpression ")";

Feature:
  "feature" (retained?="retain")? feature=ID;

OrExpression:
  operands+=AndExpression ("or" operands+=AndExpression)*;

AndExpression:
  operands+=Operand ("and" operands+=Operand)*;

Operand:
  (isNot?="not")? expression=Atom;

Atom:
  feature=ID | "(" expression=OrExpression ")";

FeatureModelImport:
  "featuremodel" importURI=STRING;
```

The code above allows you to reference a feature model file from a DSL and add feature clauses to attributes. If you want feature clauses on other elements, just add the *(featureClause=FeatureClause)?* snippet to it.

To understand what the *retain* keyword means, please see the *Retained Features* section below. For the rest of the other sections, you can ignore it.

We now have to add various things to several of the files in the language and editor. First, create a file *Extensions.ext* in the language project's *org.xtext.example.extensions* package. To this file add the following code:

```
import myDsl;

extension org::openarchitectureware::var::featureaccess::ext::utils reexport;

String featureModelUri(emf::EObject this):
  ((System)eRootContainer).featureModel.importURI;
```

Please make sure your language project has the Xpand nature by selecting, in the context menu of the project, the *Configure->Remove/Add Xpand nature* (and/or look into the *.project* file to check it's there!).

We also want to use constraint checks based on the *Check* language. To enable this feature in TMF, open the *GenerateMyDsl.mwe* workflow and add the following XML to the list of fragments of the Xtext generator. Then regenerate the language.

```
<fragment class="org.eclipse.xtext.generator.validation.CheckFragment"/>
```

In the *validation* package of your language you now find three *.chk* files. To the *MyDslCheck.chk* file add the following code:

```
import myDsl;

extension org::xtext::example::extensions::Extensions;
extension org::openarchitectureware::var::featureaccess::chk::featureconstraints;
```

```
context FeatureClause ERROR "no feature model imported":
  featureModelUri() != null;

context Feature ERROR "feature '"+feature+"' does not exist in feature model":
  getAllFeatures(featureModelUri()).contains( feature );

context FeatureAndList ERROR "one ore more features in list not found":
   getAllFeatures(featureModelUri()).containsAll( featureList );

context FeatureOrList ERROR "one ore more features in list not found":
   getAllFeatures(featureModelUri()).containsAll( featureList );

context Atom ERROR "feature '"+feature+"' does not exist in feature model":
  feature != null ? getAllFeatures(featureModelUri()).contains( feature ) : true;
```

Now let's focus on the editor. Code completion must be customized to show the features in the feature model when pressing *Ctrl-Space* for the *FeatureClause*. In the UI project, in to the class *MyDslProposalProvider* please add the following code:

```
  public void completeFeature_Feature(EObject model,
                  Assignment assignment,
                  ContentAssistContext context,
                  ICompletionProposalAcceptor acceptor)
  {
          List<String> features =
FeatureSupport.getAllFeatures(getFeatureModelUri(model));

          for(String f : features){
                  ICompletionProposal completionProposal =
                          createCompletionProposal(f, context);
                  acceptor.accept(completionProposal);
          }
  }

  private String getFeatureModelUri(EObject o){
          return ( (FeatureModelImport)

  ((org.xtext.example.myDsl.System)EcoreUtil.getRootContainer(o)).getFeatureModel
() ).getImportURI();
  }
```

Finally, let's customize the icons and the labels. To the *MyDslLabelProvider* class add this:

```
  String image( EObject ele ){
          return ele.eClass().getName().toLowerCase()+".gif";
  }
```

Also make sure you copy the icons in *org.openarchitectureware.var.featureaccess/icons* into the icons folder of your UI project (you may have to create the *icons* folder in the root of the UI project).

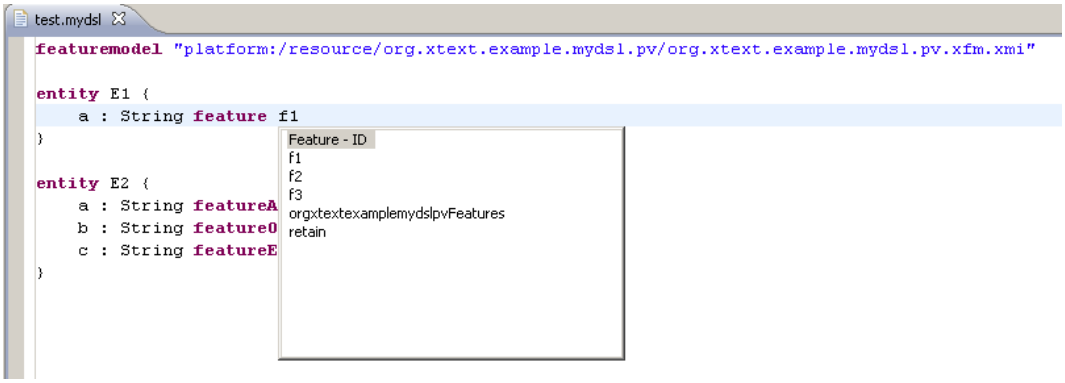
Example

After regenerating your DSL, the following should be a valid instance:

```
featuremodel
"platform:/resource/org.xtext.example.mydsl.pv/org.xtext.example.mydsl.pv.xfm.xmi
"

entity E1 {
  a : String feature f1
}

entity E2 {
  a : String featureAndList (f1, f2, f3)
```

```
  b : String featureOrList (f1, f2)
  c : String featureExp (f3 and not ( f1 or f2) )
}
```

And code completion, as well as constraint checking should also work:



Negative Variability

Create a generator

To see the result of tailoring the model create a template in the generator project created by the wizard. The example below simply dumps the model in the same way as the original syntax.

```
«DEFINE main FOR System»
  «FILE "output.txt"»
          «FOREACH entities AS e»
                  entity «e.name»  {
                          «FOREACH e.attributes AS a»
                                  «a.name» : «a.type»
                          «ENDFOREACH»
                  }
          «ENDFOREACH»
  «ENDFILE»
«ENDDEFINE»
```

In the *org.xtext.example.mydsl.generator* project add the following two plugin dependencies:
com.ps.consul.eclipse.ecore, org.openarchitectureware.var.tailor

In the generator project's *MyDslGenerator.mwe* add this meta model package to the set of registered packages:

```
  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
      <platformUri value=".."/>
          <registerGeneratedEPackage
value="com.ps.consul.eclipse.ecore.pvmodel.PvmodelPackage" />
  </bean>
```

Also, in the MweReader component replace the example model file with a workflow variable *${modelFile}*. The workflow should look like this:

```
<workflow>

  <component class="org.eclipse.emf.mwe.utils.DirectoryCleaner" directory="src-
gen"/>

  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
      <platformUri value=".."/>
          <registerGeneratedEPackage
value="com.ps.consul.eclipse.ecore.pvmodel.PvmodelPackage" />
```

```
  </bean>

  <component class="org.eclipse.xtext.MweReader" uri="${modelFile}">
          <!-- this class will be generated by the xtext generator -->
          <register class="org.xtext.example.MyDslStandaloneSetup"/>
  </component>

  <component class="org.eclipse.xpand2.Generator">
          <metaModel
class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>
          <fileEncoding value="Cp1252"/>
          <expand value="templates::Template::main FOR model"/>
          <genPath value="src-gen"/>
  </component>

</workflow>
```

(Btw, you can delete the *model* package in the generator project…)

Adapting the Example

In the *org.xtext.example.mydsl.test* project add a dependency to the *org.xtext.example.mydsl.generator* plugin and create a simple workflow stub *test.mwe*:

```
<workflow>
  <component file="workflow/MyDslGenerator.mwe"
modelFile="platform:/resource/org.xtext.example.mydsl.test/src/test.mydsl" />
</workflow>
```

You can now run this workflow. In case the workflow does not even execute, make sure that all of the dependencies of the generator plugin are reexported to make them visible to the example project.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: org.xtext.example.mydsl.generator
Bundle-Vendor: My Company
Bundle-Version: 1.0.0
Bundle-SymbolicName: org.xtext.example.mydsl.generator; singleton:=true
Eclipse-RegisterBuddy: org.eclipse.xtext.log4j
Bundle-ActivationPolicy: lazy
Require-Bundle: org.xtext.example.mydsl;visibility:=reexport,
 org.eclipse.xpand;visibility:=reexport,
 org.eclipse.xtend;visibility:=reexport,
 org.eclipse.xtext;visibility:=reexport,
 org.eclipse.emf.mwe.core;visibility:=reexport,
 org.eclipse.emf.mwe.utils;visibility:=reexport,
 org.eclipse.xtend.typesystem.emf;visibility:=reexport,
 com.ps.consul.eclipse.ecore;bundle-version="3.0.1";visibility:=reexport,
 org.openarchitectureware.var.tailor;bundle-version="1.0.0";visibility:=reexport
Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

The result of the workflow should be a file *output.txt* in the *src-gen* folder with the following content (and maybe a couple of additional empty lines).

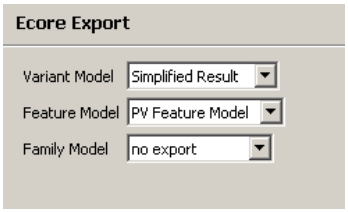
```
entity E1  {
  a : String
}

entity E2  {
  a : String
  b : String
  c : String
}
```

So far, no surpise. We didn't evaluate the feature dependencies, so the output is the same as the input, without the feature dependency clauses.

Create a Variant

In the *org.xtext.example.mydsl.pv* project, create a configuration space called *cfg*, basically a location for configuration models (*.vdm*). When creating one via the wizard, one configuration is automatically created, it is called *cfg.vdm*;

Make sure in the *org.xtext.example.mydsl.pv* project properties the following two export options are set:



Adding a Tailor step to the Generator

In the generator project's *MyDslGenerator.mwe* remove the *MweReader* component and add this:

```
 <component file="org/openarchitectureware/var/tailor/model/tailorPV.mwe"
        standaloneSetup="org.xtext.example.MyDslStandaloneSetup"
        architectureModelFile="${modelFile}"
        configurationModelUri="${configModelUri}"
        constraintFile="org::xtext::example::validation::MyDslChecks" />
        />
```

In the *org.xtext.example.mydsl.test* project you have to add a new parameter in the *test.mwe* workflow: you have to pass in the URI of the selected configuration.

```
<workflow>
 <component file="workflow/MyDslGenerator.mwe"
  modelFile="platform:/resource/org.xtext.example.mydsl.test/src/test.mydsl"
  configModelUri="platform:/resource/org.xtext.example.mydsl.pv/cfg/cfg.vdm.xmi"
  />
</workflow>
```

Testing Model Tailoring

If you now make sure that none of the features in the configuration model (*.vdm*) is selected rerun the *test.mwe* workflow, the output should look like this:

```
entity E1  {

}

entity E2  {

}
```

If you then, for example, select the feature f1 to be in the configuration, the output should look as follows:

```
entity E1  {
  a: String
}

entity E2  {
```

```
  b: String
}
```

Retained Features

All of the discussion above focused on *removing* model elements if the feature they are associated with is not selected. Consequently, a subsequent code generator cannot explicitly exploit the feature dependency – the elements are simply deleted. In many cases this is what you want. However, in some cases this is not the desired behavior; you want to be able to write a generator or transformation that uses the feature-dependency information.

So you need to be able to *retain* the feature dependency in the model until it is fed into the generator. The *retain* keyword can be used for this.

```
entity E1 {
  a : String featureAndList retain (f1, f2, f3)
  b : String featureOrList retain (f4, f5)
  c : String featureExp retain (f4 and not ( f1 or f2) );
  d : String feature retain f4;
}
```

To find out whether a feature clause (any of the forms shown above) is *true* you can include the extension file *org::openarchitectureware::var::featureaccess::ext::utils* and call the *isFeatureClauseTrue* extension, passing in the feature clause.

To make this work you have to run the remover component (as shown above) before. It is loading the actual configuration model and makes it available to the *isFeatureClauseTrue* extension.

Here is example code to verify this functionality. Update the template to call a helper function for each attribute:

```
  «FOREACH e.attributes AS a»
          «a.name» : «a.type»  «a.featureClauseValue()»
  «ENDFOREACH»
```

In the *Extensions.ext* right next to the templates implement this helper function as follows. As you can see, it calls the *isFeatureClauseTrue* utility mentioned above.

```
import myDsl;

extension org::openarchitectureware::var::featureaccess::ext::utils;

featureClauseValue( Attribute a ):
  a.featureClause.isFeatureClauseTrue();
```

Querying the configuration model directly

Instead of evaluating feature clauses expressed in the model in a transformation or generator, you can also directly query the feature model.

Here is an example:

```
  «IF isFeatureSelected("SubclassHint")»
          …
  «ENDIF»
```

Note that there's no code completion or static checks towards the feature model. The name of the feature is provided as a String.

Also, the query is run against the configuration model loaded earlier in the workflow, see the workflow example in the above section *Adding a Tailor step to the Generator*.

Aspects – Positive Variability

First we have to extend the grammar to contain pointcuts, pointcut clauses and tags. Please add the red stuff to your grammar and rebuild the language.

```
System:
  (featureModel=FeatureModelImport)?
  (entities+=Entity)*;

Entity:
  (pointcut=Pointcut)?
  "entity"
  name=ID
  (tags=TagsClause)? (featureClause=FeatureClause)? "{"
          (attributes+=Attribute)*
  "}";

Attribute:
  name=ID ":" type=ID (featureClause=FeatureClause)?;

// -------------------------------
// Feature Stuff

…

// -------------------------------
// AO Stuff

TagsClause:
  "tags" "(" (tags+=Tag)* ")";

Tag:
  name=ID;


Pointcut:
  "aspect" "{" (matches+=Match)* "}";

Match:
  AllMatch | ExactNameMatch | StartsWithNameMatch | EndsWithNameMatch | TagMatch;


AllMatch:
  "*";

ExactNameMatch:
  "name" "=" name=ID;

StartsWithNameMatch:
  "name" "startswith" name=ID ;

EndsWithNameMatch:
  "name" "endswith" name=ID;

TagMatch:
  "tag" "=" name=ID;
```

Testing Model Aspects

In the *org.xtext.example.mydsl.test* project change the *test.mydsl* file to contain the following:

```
featuremodel "platform:/resource/my.dsl.pv/my.dsl.pv.xfm.xmi"

entity E1 {
  a : String feature f1
}

entity E2 tags(t1) {
}

entity  E3 tags (t1) {
}

aspect {name startswith E} entity aoAll feature f3 {
  aAll : int
}

aspect {name=E2} entity aoE2 feature f3 {
  a2 : int
}

aspect {tag=t1} entity aoT1 feature f3 {
  aeT1 : int
}
```

Make sure that feature *f3* is selected in your configuration model. Rerun *test.mwe* to generate the output. Here is the expected result:

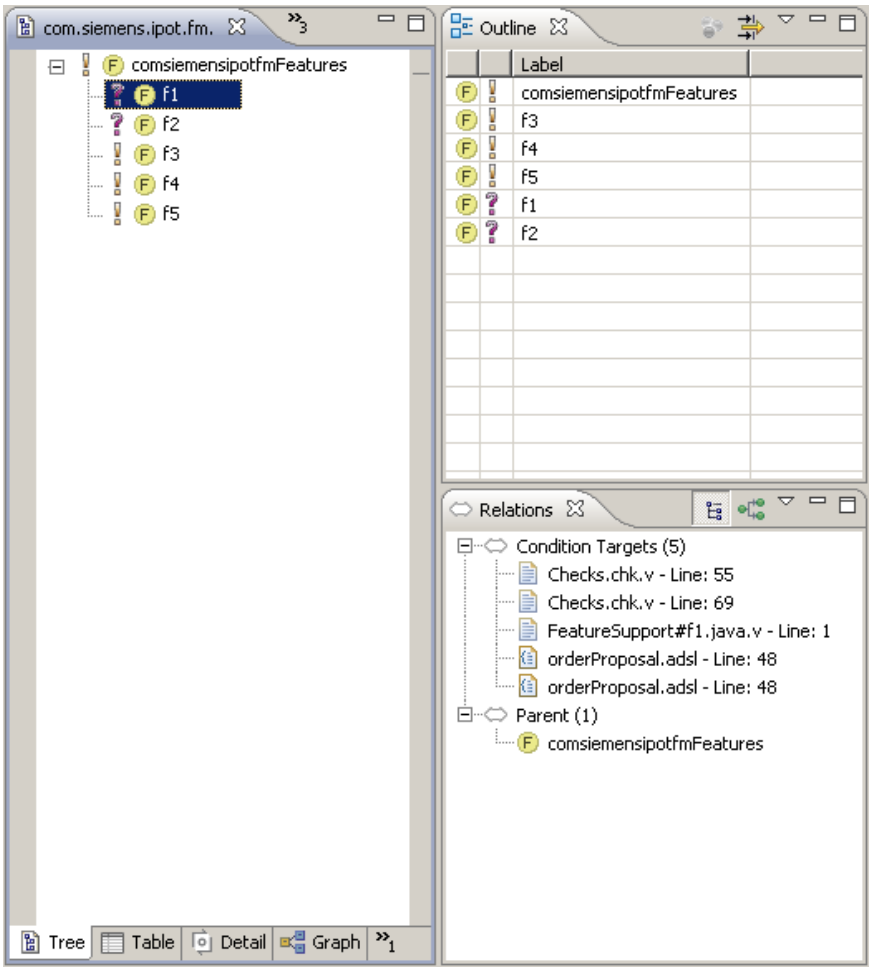
```
entity E1  {
   a : String
  aAll : int
}

entity E2  {
  aeT1 : int
  a2 : int
  aAll : int
}

entity E3  {
  aeT1 : int
  aAll : int
}
```

The *aoEAll* aspect has adviced all entities. Therefore, each of them has the *aAll* attribute in the output. The *aoE2* aspect advises only *E2*. Hence only *E2* has the *a2* attribute. Finally, *aoT1* advises all entities with the tag *t1*. Since *E2* and *E3* have this tag, they end up with the *aeT1* attribute in the output.

Note how all three aspects depend on the feature *f3*. If you remove this from the configuration and regenerate the output, no AO weaving will take place since the aspect itself will have been removed before weaving.

The Indexer

To make sure, you don't lose the overview of your feature dependencies, it is possible, to integrate the feature dependent Xtext models with pure::variants relations view.



To do this, you have to implement an extension point in pure::variants relations builder plug-in. For the feature syntax explained above, we already provide such an extension. It can be found in the following project:

org.openarchitectureware.var.pv.indexer

To implement indexing support for your own DSL (illustrated with the MyDsl example) do the following:

Create a new plugin project. Add dependencies to

- *com.ps.consul.eclipse.relationsbuilder*
- *org.eclipse.core.resources*
- *org.openarchitectureware.var.pv.indexer*

Extend the extension point *com.ps.consul.eclipse.relationsbuilder.Indexer* in the following way. Adapt the *id*, *name* and *extensions* parameter to your needs, while reusing the existing class *org.openarchitectureware.var.pv.indexer.FeatureDependencyIndexer* provided as part of the *org.openarchitectureware.var.pv.indexer* plugin.

```
   <extension
         point="com.ps.consul.eclipse.relationsbuilder.Indexer">
      <indexer
            class="
org.openarchitectureware.var.pv.indexer.FeatureDependencyIndexer"
            extensions="mydsl"
            id="org.xtext.example.mydsl.indexer.MyDslIndexer"
            name="MyDslIndexer">
```

```
    </indexer>
  </extension>
```

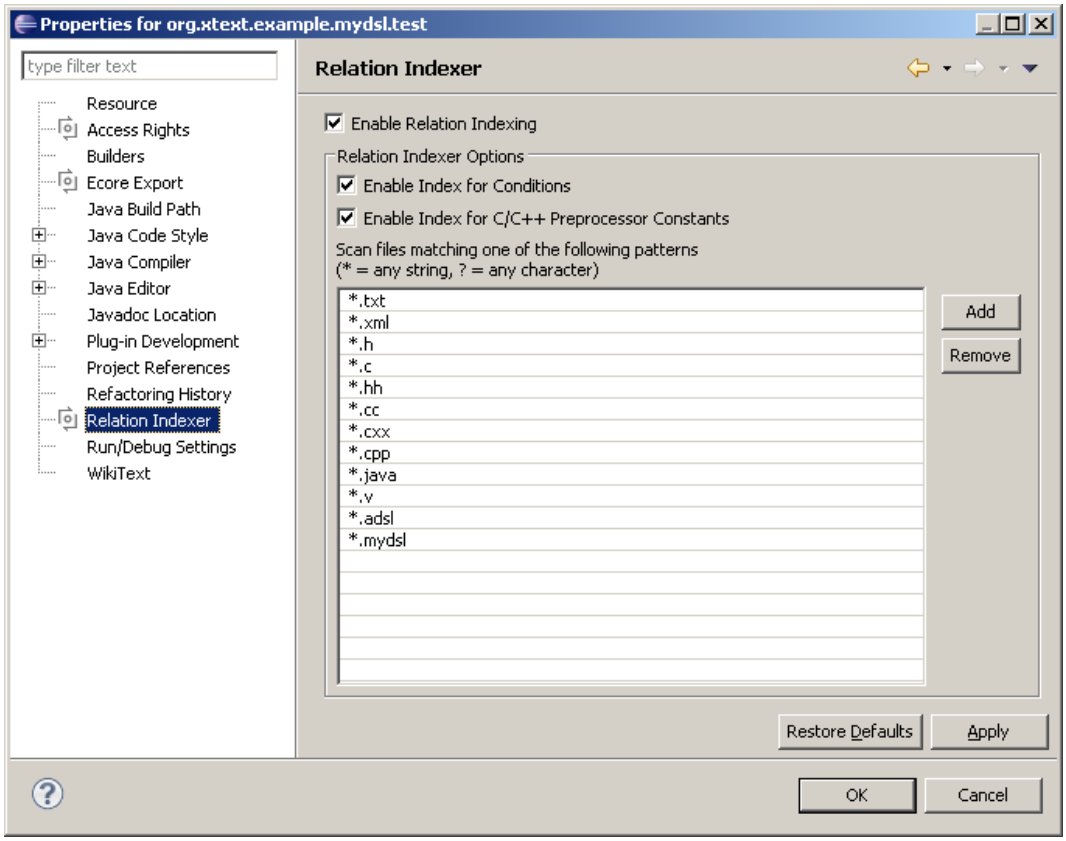
Make sure that the project that contains the files that should be indexed has the following two natures:

```
<nature>com.ps.consul.eclipse.core.ConsulNature</nature>
<nature>com.ps.consul.eclipse.relationsbuilder.RelationIndexNature</nature>
```

It also has to have the following builder:

```
<name>com.ps.consul.eclipse.relationsbuilder.RelationsBuilder</name>
```

Finally, in the indexer properties the file extension of your new to-be-indexed files is added:



This indexer reindexes *.mydsl* files whenever they are changed. It is good practice, after restarting Eclipse, to also trigger a rebuild, to clean and reestablish the index.

Adapting Text Files and Code

Whenever parts of your application are not described with models but rather with code or other textual artifacts, you need a way of also adapting them based on features. Consider a Java interface that, depending on a feature, contains a certain method or not. Then an implementation class of that interface has to provide an implementation of that method if the configuration feature is selected. If that class is manually written, you have to somehow adapt the manually written code depending on the selected features.

Notations

The way this works in principle is this: you create a *.v* file that contains all the code that might go into the manually written code. In that file, you use comments and certain notations to mark up parts of the code that depend on features. Here is an example where the second method in the class depends on the feature *f2*. The file is called *SomeModule.java.v*

```
public class SomeModule extends SomeModuleBase {

  @Override
  public OrderProposal proposeOrder(Dealer Dealer) {
          // TODO Auto-generated method stub
          return null;
  }

  //# f2
  @Override
  public void op1() {
          System.out.println( "haha" );
  }
  //~# f2

}
```

This file is typically located in a directory that contains only *.v* files. A workflow component copies those files to the actual source directory, processing the feature dependent regions on the fly. It also removes the *.v* extension. In the example above, the *op1* method will only be in the resulting Java file, if the feature *f2* is selected.

The following piece of code shows the workflow configuration:

```
  <component class="org.openarchitectureware.var.tailor.code.CoderComponent">
          <sourcePath value="platform:/resource/org.xtext.example.mydsl.test/src-
variants"/>
          <genPath value="platform:/resource/org.xtext.example.mydsl.test/src"/>
          <configurationModelSlot value="configurationModel"/>
          <configurationModelWrapperClass value="
org.openarchitectureware.var.featureaccess.pv.PVConfigurationModelWrapper"/>
  </component>
```

The *sourcePath* describes where the *.v* files can be found. The *genPath* defines where the processed files will rest. The *configurationModelSlot* is the name of a workflow slot in which the already loaded configuration model can be found (if you used the model tailoring before, this slot is *configurationModel*). Finally, the *configurationModelWrapperClass* is a strategy to work with the model. The class shown in the code above is the one you have to use with pure::variants.

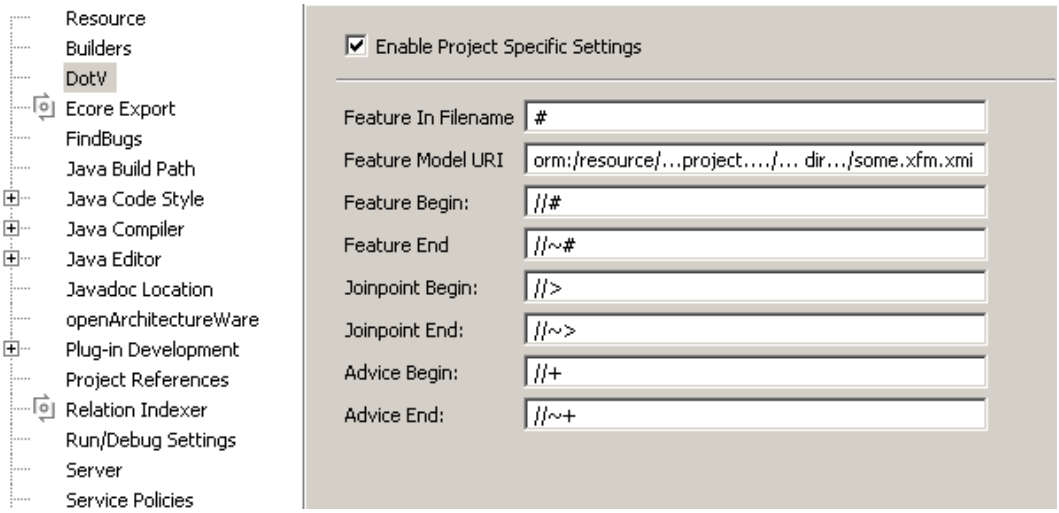
Note: if you don't use the model tailoring before, the configuration model can be loaded with the normal EMF model loading workflow component since pure::variants exports the feature model as an XMI file (obviously you have to load this exported to XMI file, not the VDM file directly).

The Builder

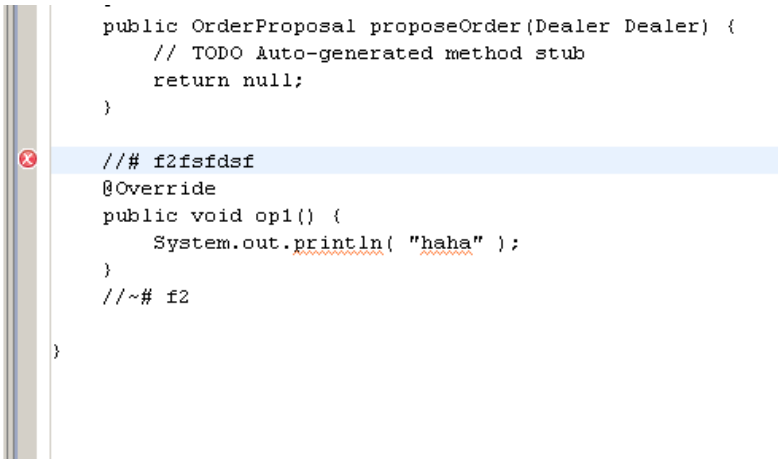
To check your markup in *.v* files against the feature model, you can use the *DotVBuilder*. If you have installed the variant management plug-ins, your context menu for projects will contain an *Add/Remove DotV Nature* entry. If you select this, the builder will be enabled. Before it does anything sensible

though, you will have to configure them feature model in the project properties.

Open your project properties and in there select the *DotV* tab.



You can change all the special characters that marked up feature dependencies, although this is not recommended. However, you have to enter the feature model URI, using the *platform:/resource* notation. You will then get an error message, if you refer to a non-existent feature.



Note that after restarting eclipse, you will have to do a full rebuild to update the cache.

The Indexer

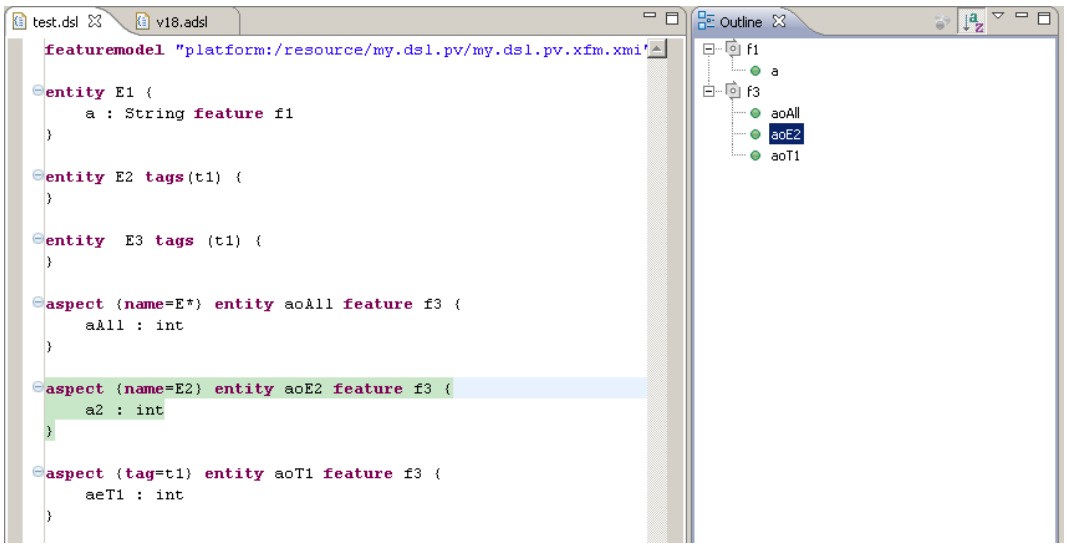
There is also a predefined indexer for out *.v* files which is active automatically if you install the variant management plug-ins. To activate it for your project open your project properties and select the Relation Indexer tab. There, add the *.v* extension to the list of extensions. Click okay and do a full rebuild.

You should now get relations into your *.v* files, just as shown above with the *mydsl* files.

Currently Not Support – Scrapheap

Adapting the outline View

You might want to adapt the outline view so it provides a viewpoint that puts the feature on top, and then arranges below it all the model elements that depend on this feature.



Here is the code:

```
viewpoints(): {
  "Features"
  };

viewpointIcon(String vpName) :
  switch (vpName) {
          case "Features": "vp_feature.gif"
          default: "vp_default.gif"
  };


// ------------------------------------------------------------------------
// Features

create UIContentNode outlineTree_Features(emf::EObject model) :
  let features =
model.allLocalElements().typeSelect(FeatureClause).collect(fc|fc.feature).toSet()
:
          setLabel(model.label()) ->
          setImage(model.image()) ->
          setContext(model)->
          children.addAll( features.createFeatureNode(model) );

create UIContentNode createFeatureNode( String feature, emf::EObject model ):
  setLabel(feature) ->
  setImage("feature.gif") ->
  setContext(null)->
  children.addAll( feature.createRefNodes(model) );

createRefNodes( String feature, emf::EObject model ):
  model.allLocalElements().typeSelect(FeatureClause).select(fc|fc.feature ==
feature).collect(fc|fc.eContainer).createRefNode();
```

```
create UIContentNode createRefNode( emf::EObject obj ):
  setLabel(obj.label()) ->
  setImage(obj.image()) ->
  setContext(obj);
```

You might have to copy the *feature.gif* and the *vp_feature.gif* from the *org.openarchitectureware.var.featureaccess* plugin's *icons* folder over into your language editor's *icons* folder.