Variability in Xtext DSLs

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.

Feature Access

Create an Xtext Project

The 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.xtxt* file as the language grammar.

```
System:
  (entities+=Entity)*;

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

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

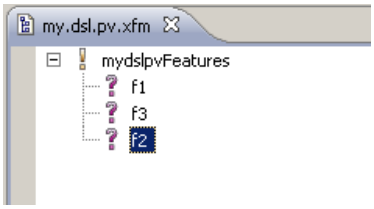
Generate the DSL, and verify that it works. Create an openArchitectureWare project *my.dsl.test* and put a *test.dsl* file with the following content into the *src* folder.

```
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 *my.dsl.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 (*my.dsl.pv.xfm*) create a couple of features. Make sure these are optional features!



Then save the model. A *my.dsl.pv.xfm.xmi* next to *my.dsl.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. Because of the missing language modularization features of Xtext 4.x this cannot be provided as a "library" of some form. Copy and Paste is necessary.

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

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

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

FeatureClause:
  "feature" feature=ID;

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

The code above allows 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.

We now have to add various stuff to several of the files in the language and editor. Let's start with the language utilities in *Extensions.ext*. To the *Extensions.ext* add this, making sure that you replace the *System* with your top level meta class.

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

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

The constraints, *Checks.chk* must also be extended, to detect missing feature model imports and references to undefined features.

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

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

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*. To *ContentAssist.ext* add this:

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

completeFeatureClause_feature(emf::EObject ctx, String prefix) :
  proposeFeatures(ctx, prefix);

proposeFeatures(emf::EObject ctx, String prefix) :
  let features = getAllFeatures(ctx.featureModelUri()):
          let filteredFeatures = (prefix != null ?
features.select(f|f.startsWith(prefix)) : features) :
                  filteredFeatures.collect(ff|newProposal( ff, ff,
"featureclause.gif") );
```

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

```
image( FeatureClause this ): "featureclause.gif";
image( FeatureModelImport this ): "featuremodelimport.gif";
label( FeatureClause this ): feature;
label( FeatureModelImport this ): "feature model "+uri;
```

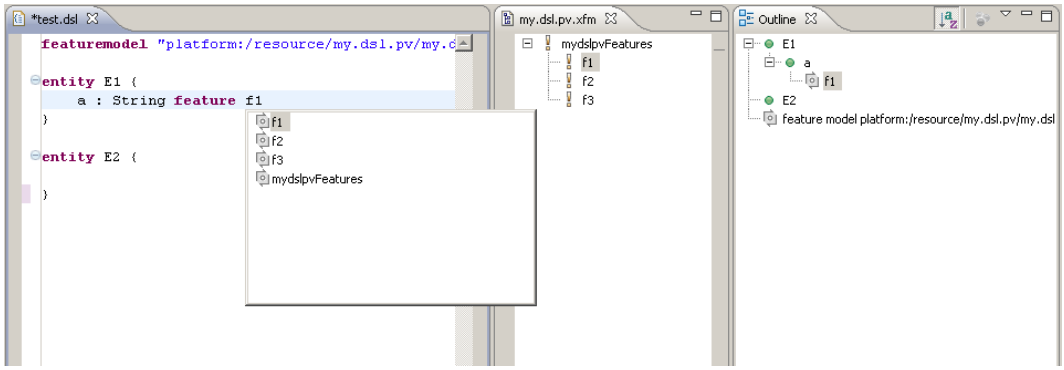
Also make sure you copy the icons in *org.openarchitectureware.var.featureaccess/icons* into the icons folder of your *my.dsl.editor*

Example

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

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

entity E1 {
  a : String feature f1
}

entity E2 {

}
```

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 generator 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 *my.dsl.generator* project add the following two plugin dependencies: *com.ps.consul.eclipse.ecore* and *org.openarchitectureware.var.tailor*

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

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

Adapting the Example

In the *my.dsl.test* project add a dependency to the *my.dsl.generator* plugin and create a simple workflow stub *test.oaw*:

```
<workflow>
  <component file="org/example/dsl/generator.oaw" modelFile="test.dsl" />
</workflow>
```

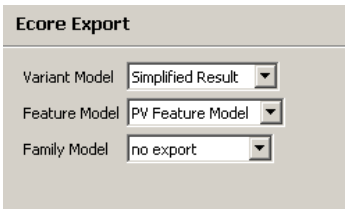
You can now run this workflow; it should generate the following code. No surprise.

```
entity E1  {
  a : String
}

entity E2  {
}
```

Create a Variant

In the *my.dsl.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 *my.dsl.pv* project properties the following two export options are set:



Adding a Tailor step to the Generator

In the generator project's *generator.oaw* remove the parser component and add this:

```
 <component file="org/openarchitectureWare/var/tailor/model/tailorPV.oaw"
        architectureModelFile="${modelFile}"
        configurationModelUri="${configModelUri}"
        dslPackage="platform:/resource/my.dsl/src-
gen/org/example/dsl/mydsl.ecore"
        parserCartridge="org/example/dsl/parser/Parser.oaw"
        constraintFile="org::example::dsl::Checks" />
        />
```

In the *my.dsl.test* project you have to add a new parameter to the invocation of the *generator.oaw* workflow: you have to pass in the URI of the selected configuration.

```
<workflow>

 <component file="org/example/dsl/generator.oaw"
        modelFile="test.dsl"
        configModelUri="platform:/resource/my.dsl.pv/cfg/cfg.vdm.xmi"
        />

</workflow>
```

Testing Model Tailoring

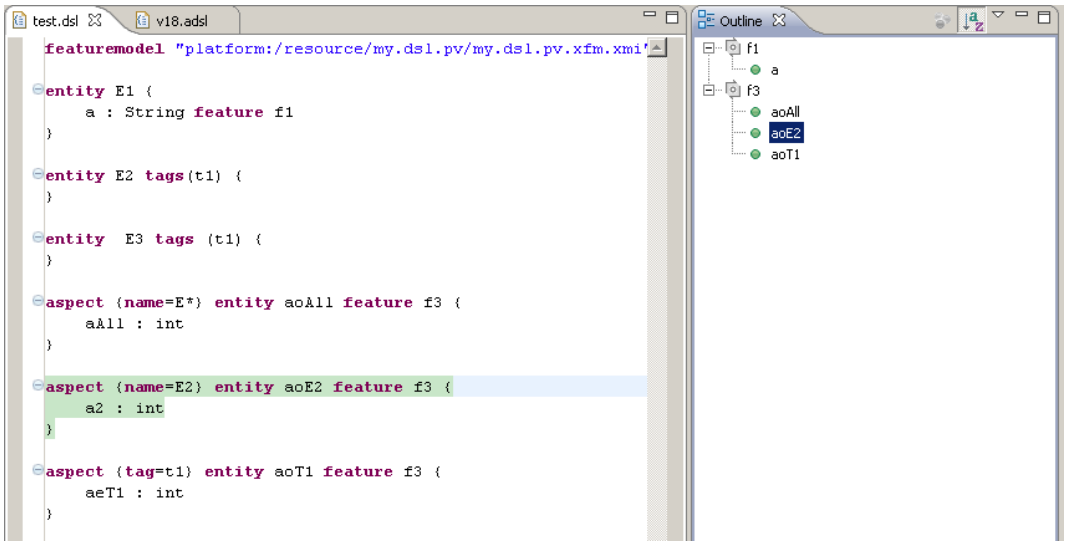
If you now make sure the feature *f1* is not selected in the *cfg.vdm* configuration model and rerun the *test.oaw* workflow, the output should look like this:

```
entity E1  {

}

entity E2  {

}
```

Note how the attribte *a* is missing, because it's feature has been unselected.

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.

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

FeatureClause:
  "feature" feature=ID;

FeatureModelImport:
  "featuremodel" uri=STRING;


// -------------------------------
// 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" "=" name=ID "*";

EndsWithNameMatch:
  "name" "="  "*" name=ID;

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

Testing Model Aspects

In the *my.dsl.test* project change the *test.dsl* 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=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
}
```

Rerun *test.oaw* to generate the output. Here is the expected result:

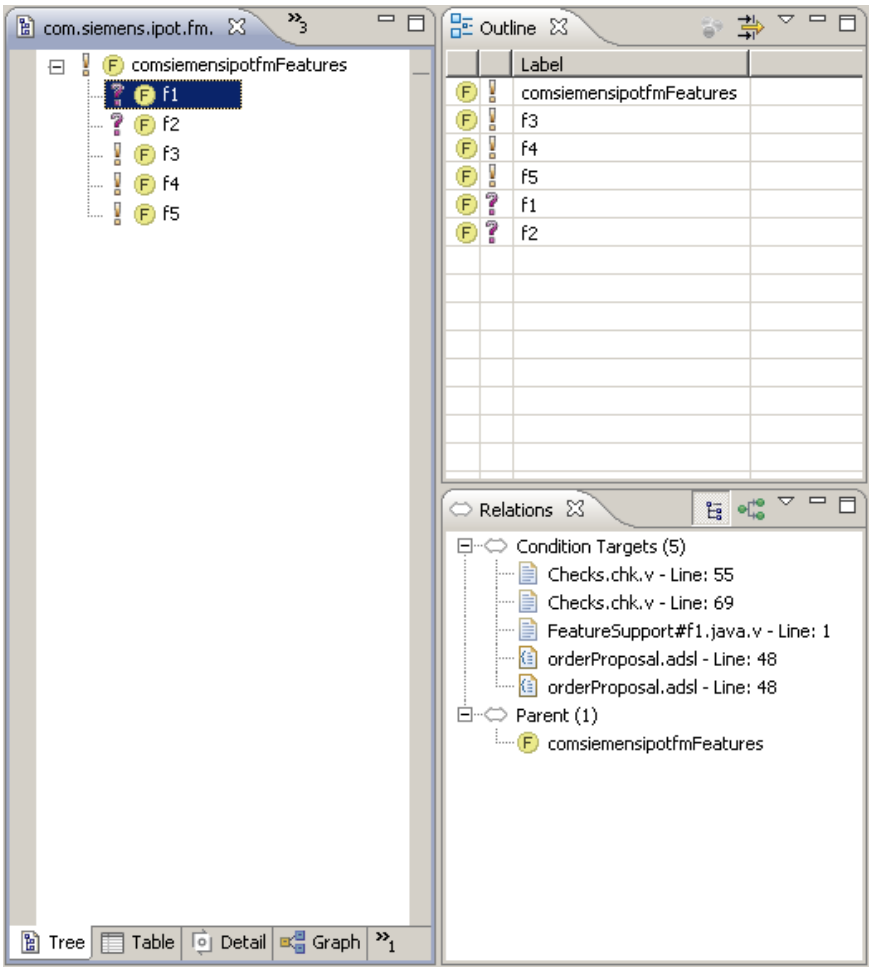
```
entity E1  {
   a : String
  aAll : int
}

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.adsl

For your reference, and as an example from which to copy for your own extensions, here is the implementation code, the plug-in manifest and the plugin.xml.

Implementation:

```
package org.openarchitectureware.var.pv.relbuilderadsl;

public class AdslFileIndexer implements IIndexer {

 public static final String FEATURECONSTANT = "feature";

 @Override
 public void scan(IFile file, ITargetCache cache) throws CoreException {
        InputStream i = null;
        try {
                i = file.getContents();
                BufferedReader br = new BufferedReader(new
InputStreamReader(i));
                int lineCount = 0;
                while (br.ready()) {
                        lineCount++;
                        String l = br.readLine();
                        parseLine(l, lineCount, file, cache);
                }
                br.close();
        } catch ( IOException ex ) {
```

```
                Status s = new Status(IStatus.ERROR, Activator.PLUGIN_ID, 0,
ex.getMessage(), ex);
                CoreException coreException = new CoreException(s);
                throw coreException;
        } finally {
                try {
                        i.close();
                } catch (IOException ignore) {
                }
        }
 }

 private void parseLine(String l, int lineNo, IFile file, ITargetCache cache) {
        StringTokenizer st = new StringTokenizer(l, " \t;");
        while (st.hasMoreElements()) {
                String token = (String) st.nextToken();
                if ( token.equals(FEATURECONSTANT)) {
                        if ( st.hasMoreTokens() ) {
                                String featureName = st.nextToken();
                                cache.add(featureName, file, lineNo);
                        }
                }
        }

 }
}
```

Manifest:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Relbuilderadsl Plug-in
Bundle-SymbolicName: org.openarchitectureware.var.pv.indexer.adsl;singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: org.openarchitectureware.var.pv.relbuilderadsl.Activator
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.core.resources;bundle-version="3.4.0",
 com.ps.consul.eclipse.relationsbuilder;bundle-version="3.0.2"
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="com.ps.consul.eclipse.relationsbuilder.Indexer">
      <indexer

class="org.openarchitectureware.var.pv.relbuilderadsl.AdslFileIndexer"
            extensions="adsl,foo,bar"
            id="org.openarchitectureware.var.pv.relbuilderadsl.AdslFileIndexer"
            name="AdslFileIndexer">
      </indexer>
   </extension>

</plugin>
```

This indexer reindexes ADSL 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 pots of your application are not described with models but rather with code or all the 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 off 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. An oAW 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/…project…/…directory…"/>
  <genPath value=" platform:/resource/…project…/…directory…"/>
  <configurationModelSlot value="…configurationModel…"/>
   <configurationModelWrapperClass
value="org.openarchitectureware.var.featureaccess.pv.PVConfigurationModelWrapper"
/>
</component>
```

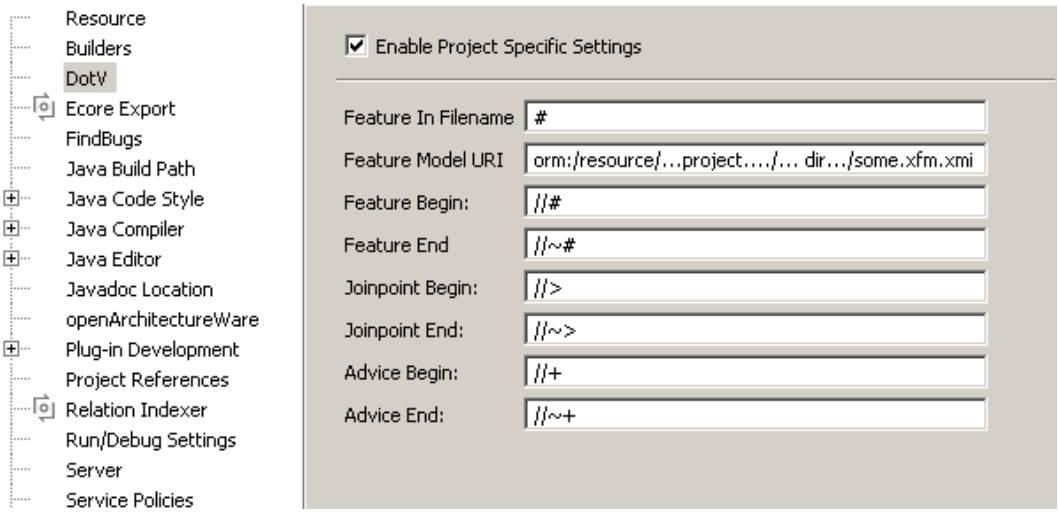
The sourcePath describes where the .v files can be found. The genPath defines where the processed files will rest. The configuration model slot is the name of a workflow slot in which the already loaded configuration model can be found. Finally, the configurationModelWrapperClass is a strategy to work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

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).

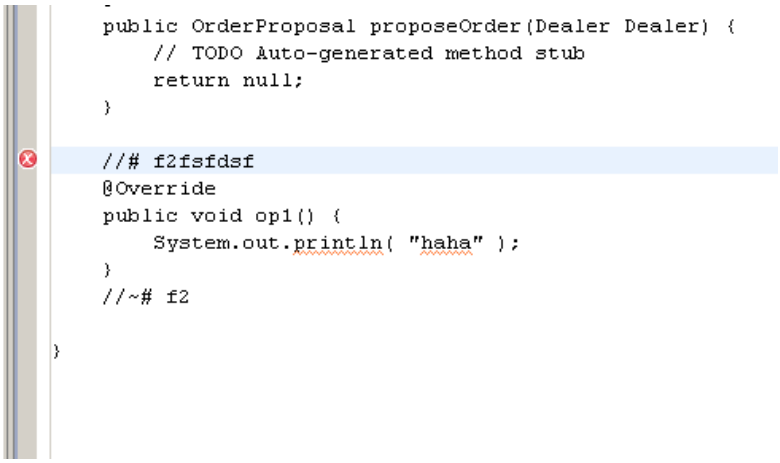
The Builder

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 a "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.



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

    //# f2fsfdsf
    @Override
    public void op1() {
        System.out.println( "haha" );
    }
    //~# f2

}
```

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

The Indexer

Just like the indexer fault the ADSL files above, 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.D. files, just as shown above with the ADSL files.

Injecting into Text Files