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.

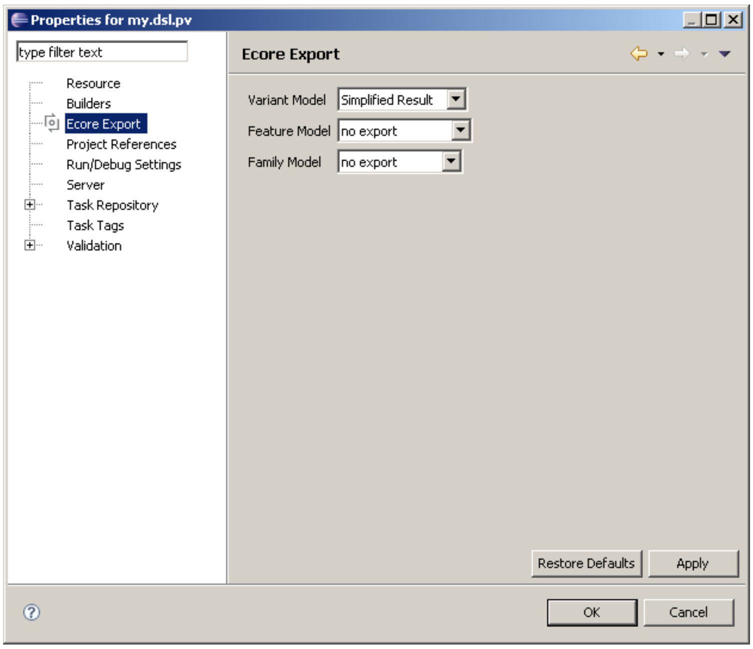
```
System:
  (entities+=Entity)*;

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 {

}
```

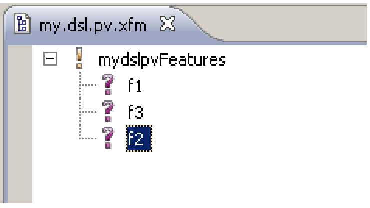
Create a p::v Project

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 Variant Model Ecore Export to *Simplified Result*.



Variant Model: Simplified Result
Feature Model: no export
Family Model: no export

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!



```
mydslpvFeatures
    f1
    f3
    f2
```

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. 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:

```
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;
```

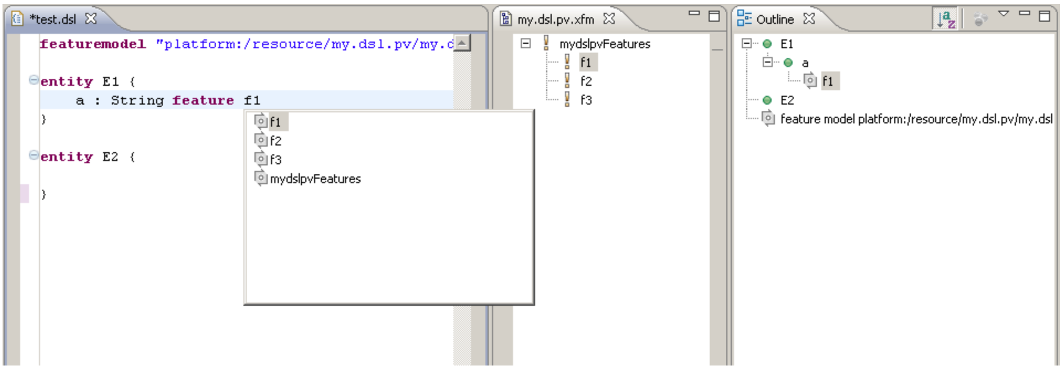
Also make sure you copy the icons in *org.openarchitectureware.var.featureaccess/icons* into the icons folder of your *my.dsl.editor*

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:



```
featuremodel "platform:/resource/my.dsl.pv/my.d

entity E1 {
    a : String feature f1
}

entity E2 {

}
```

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>
```

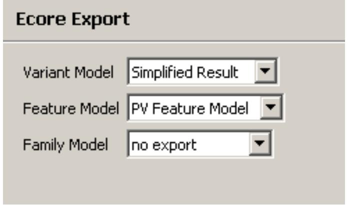
You can now run this workflow; it should generate the following code. No surprise.

```
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:



Ecore Export

Variant Model: Simplified Result

Feature Model: PV Feature Model

Family Model: no export

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

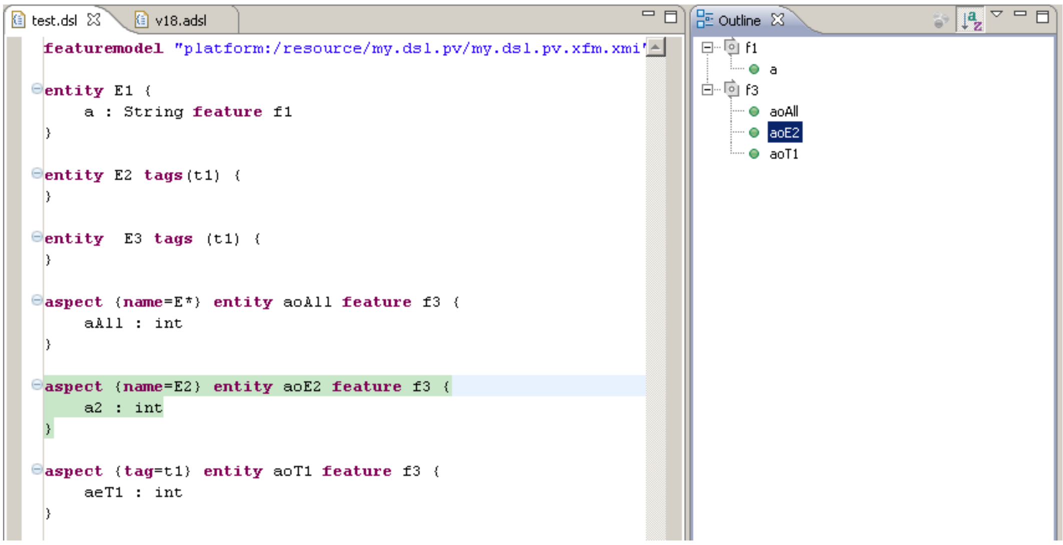
If you now make sure the feature *f1* is not selected in the *cfg.vdm* configuration model and rerun the *test.oaw* workflow, the output should look like this:

```
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.



```
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
}
```

```
f1
    a
f3
    aoAll
    aoE2
    aoT1
```

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.