VML*: A Generative Infrastructure for Variability Management Languages

Steffen Zschaler
Lancaster University, UK

September 17, 2009

Abstract

This document explains how to install and use VML*, a family of languages for variability management.

1 Introduction

A *software product line* (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. Apart from sharing a common set of features, every system also has features that are specific to this system and not shared with other systems in the SPL. An important part of managing the features of a product line and the individual systems (often called *products*) is to model the available features and their dependencies (e.g., if feature `A` is selected, feature `B` also must be selected) in an abstract form, called a feature model (e.g., [6]). While these models express what features there are and what products can be formed from them, they do not express how a specific feature is realised and, thus, how any specific product is realised. Feature models are strictly problem-space models. The realisation of the product line is designed in a different set of models using a different set of modelling languages.

This leads to a *mapping problem:* For each feature in a feature model we need to identify and specify the solution-space models and model elements associated with it to be able to systematically construct products given a selection of features. A number of different approaches for such mappings have been proposed [2, 4, 5, 7]. Most of the approaches propose a generic modelling language for expressing simple relations between features and model elements. In contrast, our VML* approach [7] proposes to construct customised languages for each solution-space modelling language used in an SPL. The main benefit of such an approach is the ability of providing more sophisticated mapping relations that have been custom designed for the specific solution-space modelling language. For example, if activity diagrams are used as part of the solution-space models, we can provide a mapping that merges additional steps into an activity if a certain feature has been selected. For class diagram models we may provide a mapping relation that introduces package-merge dependencies in a model. A more detailed discussion of the respective benefits and drawbacks of these approaches can be found in [7].

All of these languages are relatively similar. Still, without support for reuse, it can be tedious and error prone to develop the support infrastructure for a new VML language. As a solution, we developed VML*, which is a domain-specific metamodelling language for VML languages. This document describes how to use VML* to define new VML languages.

2 Installing VML*

VML* has been realised as a set of plugins for the Eclipse IDE. To work properly, it requires a number of other plugins (and their dependencies) to be installed and to be working correctly:

1. *The Textual Editing Framework (TEF).* This can be obtained from `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html`.

2. *The Epsilon Model Management Platform.* This can be obtained from `http://www.eclipse.org/gmt/epsilon/`.

3. *openArchitectureWare.* This can be obtained from `http://www.openarchitectureware.org/`.

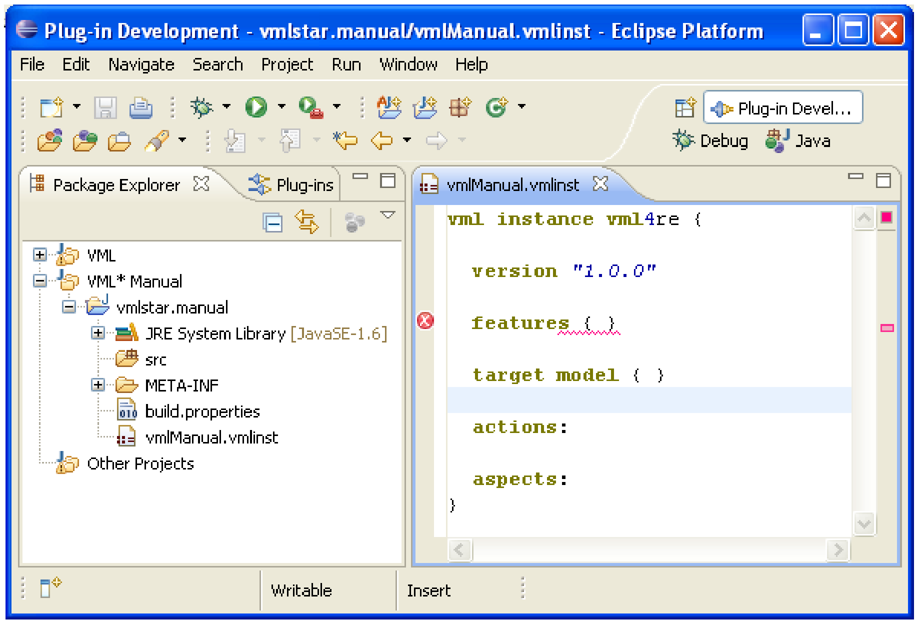
4. *Eclipse Modelling Framework.* This can be obtained from `http://www.eclipse.org/modeling/emf/`.

5. *AMPLE Tracing Framework (ATF).* This can be obtained from the AMPLE project website.

VML* itself, then, consists of four plugins, all of which need to be copied into the 'dropins' folder of your Eclipse installation (while Eclipse is not running):

- `org.ample.vmlstar.langinst,`
- `org.ample.vmlstar.langinst.model,`
- `org.ample.vmlstar.langinst.model.edit`, and
- `org.ample.vmlstar.util`

Once they are there, start Eclipse, using the '-clean' command-line option to ensure Eclipse recreates its cashed list of installed plugins. Note, you only need this command-line option when starting Eclipse for the first time after making any changes to the installed plugins. After that Eclipse can again be started without command-line options. You should now have a special editor for files with a `vmlinst` extension. Such files should also show an additional, VML*-related menu-option in their context menu.



```
vml instance vml4re {

  version "1.0.0"

  features { }

  target model { }

  actions:

  aspects:
}
```

Figure 1: A basic VML* language-instance descriptor opened in an Eclipse editor

3 Defining Variability Management Languages

To define a new VML language, we have to write a so-called language-instance descriptor as well as, possibly, a number of helper files. All of these files go into a new project, ideally an openArchitectureWare plugin project. While the language-instance descriptor can be stored anywhere in this project, the helper files must reside in a source folder, so that they will later be exported as part of the classpath made available by this plugin.

A language-instance descriptor is a text file with a vmlinst file extension. Figure 1 shows a screenshot of Eclipse with a basic VML* project and a basic language-instance descriptor open. It can be seen that the language-instance descriptor consists of a series of sections:

1. An optional *version* section defines the version number of all generated plugins.
2. The *features* section provides information about the metamodel for the feature models with which the new VML language will integrate. As we have currently not provided any of the mandatory information within this section, VML* shows an error marker. Note that usual only the first parsing error will be highlighted in this way.
3. The *target model* section analogously provides information about the metamodel of the target models with which the new VML language will integrate. Again,

further mandatory information must be provided in this section.

4. The *actions* section defines the available actions and their syntax. Actions are used in VML specifications for describing the modifications of the target model in response to selected features. The set of available actions is the major variation point between different VML languages.

5. Finally, the *aspects* section defines the different evaluation aspects that the new VML language should support. In effect, this defines the semantics of the new language for each type of evaluation.

Additionally, there are two more optional sections—*bundles* and *extensions* (not shown in Fig. 1 which can be used to import other plugins and additional model-transformation code. We will show examples of these sections as we need them. We will walk through these sections in more detail in the following.

3.1 Version Definition

The `version` section is an optional section to define the version number used for all plugins generated from the language-instance descriptor. The version number is given as a parenthesised string after the `version` keyword (cf. Fig. 1).

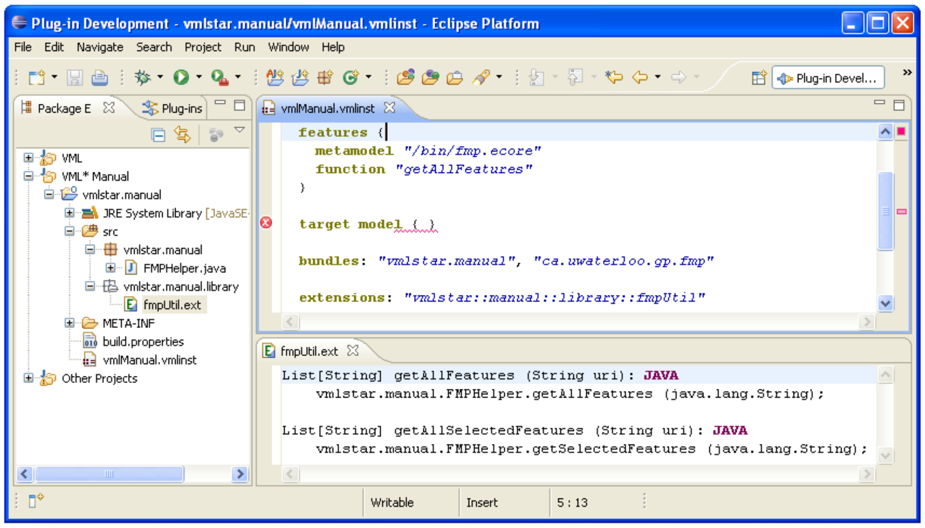
3.2 Feature-Model Definition

The `features` section is a mandatory section of every VML language-instance descriptor. It contains all information required for accessing the feature models supported by the new VML language. The section has the following elements (Figure 2 shows an example):

metamodel This element gives the path to an ecore file containing the metamodel for the feature models supported by the new language. This file must be on the classpath of any of the generated projects for this new language. To achieve this, you may have to add the plugin defining the metamodel in the `bundles` section of the language-instance descriptor. In Fig. 2 we specify usage of the FMP metamodel [3] and also include its defining bundle `ca.uwaterloo.gp.fmp` in the references.[1]

function This element references an openArchitectureWare xTend function that extracts all features from a feature model. The lower half of Fig. 2 shows such a function for FMP. (Note that for this case the actual function has been implemented in Java, because this was easier for the somewhat difficult to follow way in which FMP models are encoded.) The function is defined in a separate oAW extension file (`fmpUtil.ext`), which must be referenced from the language-instance descriptor using the `extensions` section. This extension must further

[1]The somewhat strange path `/bin/fmp.ecore` is due to a packaging error in the FMP plugin.



```
features {
  metamodel "/bin/fmp.ecore"
  function "getAllFeatures"
}

target model { }

bundles: "vmlstar.manual", "ca.uwaterloo.gp.fmp"

extensions: "vmlstar::manual::library::fmpUtil"
```

```
List[String] getAllFeatures (String uri): JAVA
    vmlstar.manual.FMPHelper.getAllFeatures (java.lang.String);

List[String] getAllSelectedFeatures (String uri): JAVA
    vmlstar.manual.FMPHelper.getSelectedFeatures (java.lang.String);
```

Figure 2: Example of a feature-model definition

be available on the classpath, which means that the defining plugin project must be listed in the `bundles` section.[2]

oAW As an alternative to `function`, the `oAW` element can be used to provide the xTend code directly in the language-instance descriptor. This is, however, no longer recommended.

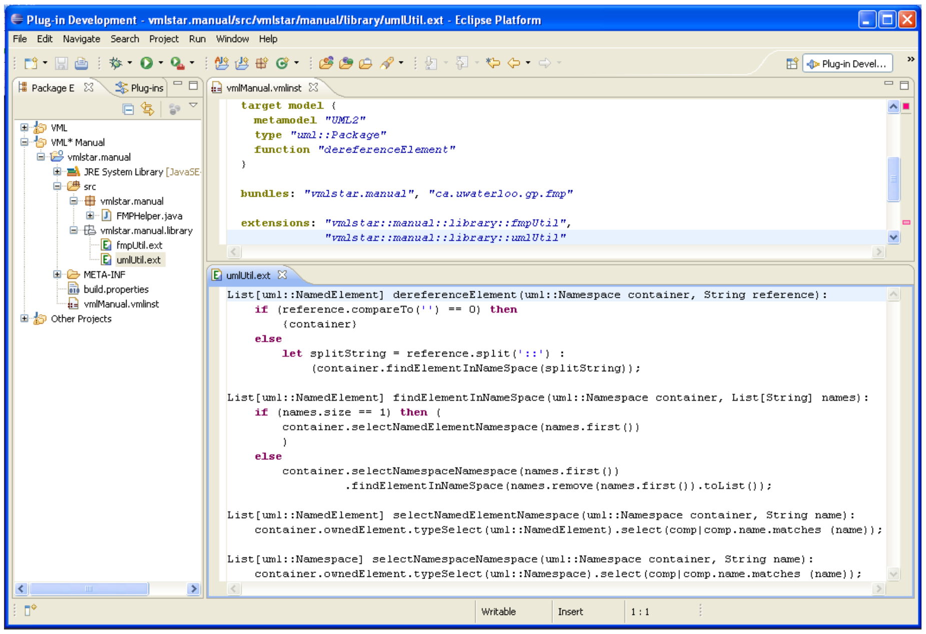
3.3 Target-Model Definition

Similarly to `features` the mandatory `target model` section defines the metamodel of target models and provides an adapter for accessing model elements in a target model. It contains the following elements (Refer to Fig. 3 for an example):

metamodel This element gives the path to an ecore file containing the metamodel for the feature models supported by the new language. This file must be on the classpath of any of the generated projects for this new language. To achieve this, you may have to add the plugin defining the metamodel in the `bundles` section of the language-instance descriptor. If the target models are UML models, instead of a path to an ecore file, this element contains the reserved string ``` ``UML2'' ```.

type This element provides the fully qualified name of the type of the top-level element of any target model. Individual name elements are separated using two colons (i.e., ‘::’).

[2]In defining this extension, we have also added the openArchitectureWare nature to our project, so that xTend syntax is automatically checked.



```
target model {
  metamodel "UML2"
  type "uml::Package"
  function "dereferenceElement"
}

bundles: "vmlstar.manual", "ca.uwaterloo.gp.fmp"

extensions: "vmlstar::manual::library::fmpUtil",
            "vmlstar::manual::library::umlUtil"
```

```
List[uml::NamedElement] dereferenceElement(uml::Namespace container, String reference):
    if (reference.compareTo('') == 0) then
        {container}
    else
        let splitString = reference.split('::') :
            (container.findElementInNameSpace(splitString));

List[uml::NamedElement] findElementInNameSpace(uml::Namespace container, List[String] names):
    if (names.size == 1) then (
        container.selectNamedElementNamespace(names.first())
        )
    else
        container.selectNamespaceNamespace(names.first())
                 .findElementInNameSpace(names.remove(names.first()).toList());

List[uml::NamedElement] selectNamedElementNamespace(uml::Namespace container, String name):
    container.ownedElement.typeSelect(uml::NamedElement).select(comp|comp.name.matches (name));

List[uml::Namespace] selectNamespaceNamespace(uml::Namespace container, String name):
    container.ownedElement.typeSelect(uml::Namespace).select(comp|comp.name.matches (name));
```

Figure 3: Example of a target-model definition

function This element references an xTend function that takes a model (of type `type`) and a textual reference and returns a list of model elements that this reference refers to. This function resides in a separate xTend extension file. Figure 3 shows an example.

oAW As an alternative to `function`, the `oAW` element can be used to provide the xTend code directly in the language-instance descriptor. This is, however, no longer recommended.

3.4 Action Definition

The `actions` section defines the syntax of the actions available for the new VML language. For each action, this includes the name of the action and a list of parameter types for this action. For example, an action called `createUseCase` with a parameter of type `String` (the name of the new use case) and a parameter of type `uml::Package` (the package in which to create the use case) would be defined like this:

```
createUseCase {
  params "String" "uml::Package"
}
```

Parameter types can be any of the following:

- *String.* This is the only basic type allowed. The action parameter can only be a simple string enclosed in double quotation marks. If other basic parameter types are required, they should be converted from a `String` parameter.
- *Target model types.* Any type from the target model metamodel can be used as a parameter type. It has to be specified using its fully qualified name (using '::' to separate name elements). An actual parameter can be given using any pointcut specification over the target model. However, only one matching model element will be randomly selected.
- *List types.* Parameter types of the form `List[type]`, where `type` stands for an arbitrary target model element type, can be used to indicate that the parameter can be replaced by a set of model elements from the target model. These can be specified by an arbitrary pointcut definition over the target model.

3.5 Semantics Definition

Finally, the section called `aspects` is used to define the different semantic aspects of the new language. Currently, only two aspects can be defined:

1. *Transformation.* This aspect defines semantics for deriving products based on a VML specification and a product configuration (i.e., a selection of features for a specific product).
2. *Tracing.* This aspect defines semantics for deriving trace links from a VML specification as part of product derivation.

The following two subsections explain the details of these two semantic aspects.

3.5.1 Transformation Aspect

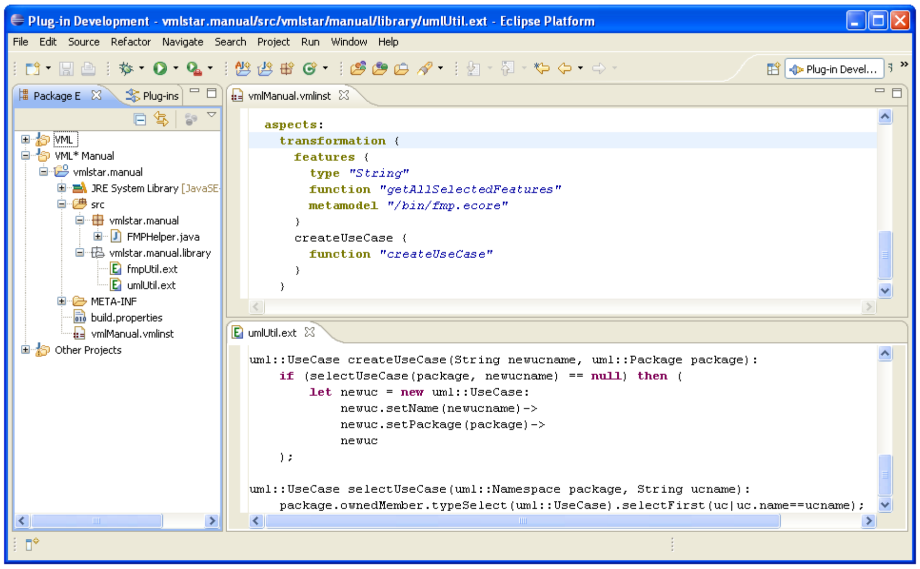
Product-derivation semantics is defined in two parts as shown in Fig. 4:

1. An adapter for accessing product configurations, and
2. for each action a piece of model-transformation implementing this actions as a product derivation step.

The first part is defined in the `features` section. This defines the following elements:

type This element provides the fully qualified name of the type of the top-level element of a product configuration. Individual name elements are separated using two colons (i.e., '::'). Alternatively, if product configurations are not provided in an ecore model, or if you want full control over reading and parsing the product-configuration file for some other reason, you can use `String` as the type.

function This element references an oAW xTend function that extracts all selected features from a feature model. The lower part of Fig. 2 shows the function referenced here. It takes one parameter of the type defined in the `type` section and returns a list of feature names.



```
aspects:
  transformation {
    features {
      type "String"
      function "getAllSelectedFeatures"
      metamodel "/bin/fmp.ecore"
    }
    createUseCase {
      function "createUseCase"
    }
  }
```

```
uml::UseCase createUseCase(String newucname, uml::Package package):
    if (selectUseCase(package, newucname) == null) then (
        let newuc = new uml::UseCase:
            newuc.setName(newucname)->
            newuc.setPackage(package)->
            newuc
    );

uml::UseCase selectUseCase(uml::Namespace package, String ucname):
    package.ownedMember.typeSelect(uml::UseCase).selectFirst(uc|uc.name==ucname);
```

Figure 4: Example transformation-aspect definition

oAW As an alternative to `function`, the `oAW` element can be used to provide the xTend code directly in the language-instance descriptor. This is, however, no longer recommended.

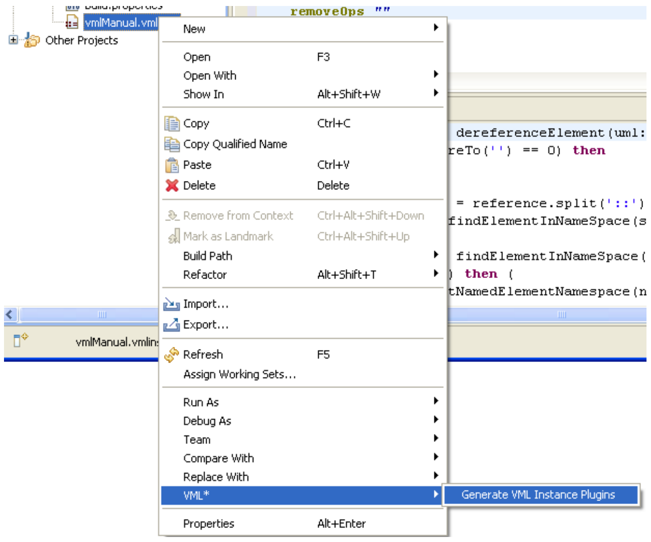
metamodel This optional element gives the path to an ecore file containing the metamodel for configuration models. This file must be on the classpath of any of the generated projects for this new language. To achieve this, you may have to add the plugin defining the metamodel in the `bundles` section of the language-instance descriptor.

The semantic of actions is defined in one simple section per action. This section only contains one element `function` referencing an oAW xTend function that implements the action. The parameter types of this function are given by the parameter types of the action. However, any list-type parameters will be unwrapped, invoking the function once for each element of the list. If there are more than one list-type parameters, the function will be invoked once for every combination of list elements.

3.5.2 Tracing Aspect

The tracing aspect defines how to derive trace links from a VML specification. It depends on the transformation aspect. It is specified using three elements:

createOps This references a number of functions in the transformation semantics code that create target model elements. These functions must return the newly created element. The references to the functions can be arbitrary valid xTend pointcuts.



```
removeOps ""
 dereferenceElement(uml:
reTo('') == 0) then
 = reference.split('::')
findElementInNameSpace(s
 findElementInNameSpace(
) then (
tNamedElementNamespace(n
```

Figure 5: Generating a VML language's infrastructure

removeOps This references a number of functions in the transformation semantics code that delete target model elements. These functions must take the element to be deleted as their first parameter. The references to the functions can be arbitrary valid xTend pointcuts.

getName This references a function that takes an `EObject` from the target model and returns a `String` that can be used as a reference to this `EObject` in a trace-link repository.

3.6 Generating the Language Infrastructure

Once a language-instance descriptor and its supporting xTend files have been written, the language-support infrastructure can be generated. To this end, right-click on the language-instance descriptor file in the navigator view and select `VML*/Generate VML Instance Plugins` as shown in Fig. 5. This will generate three plugin projects containing the complete infrastructure code for your language. You can now use your language by deploying these plugins (possibly together with any plugins you referenced in the `bundles` section).

4 Conclusions

This document presented VML* and explained how to use it to create a new VML language. We have discussed the motivation behind VML*, how to install it, and each step required to define a new VML language. For examples of such languages, please refer to similar documentation provided for VML4RE and VML4Arch provided from the AMPLE web site.

Acknowledgments

VML* and this documentation have been produced in the context of the European FP6 STREP AMPLE.

References

[1] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[2] Kryzstof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proc. 4th Int'l Conf. on Generative Programming and Component Engineering (GPCE'2005)*, pages 422–437, 2005.

[3] Generative Software Development Group, U. Waterloo. Feature modelling plugin (FMP) for Eclipse. http://gsd.uwaterloo.ca/projects/fmp-plugin/.

[4] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Proc. 12th Int'l Conf. on Software Product Lines (SPLC'08)*, pages 139–148. IEEE, 2008.

[5] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping features to models. In *Companion 30th Int'l Conf. on Software Engineering (ICSE'2008)*. ACM, 2008.

[6] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-0211990, Software Engineering Institute, 1990.

[7] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML* – a family of languages for variability management in software product lines, 2009. Submitted to SLE 2009.