# Handling Variability in Model Transformations and Generators

Markus Voelter[1], Iris Groher[2]
*[1] Independent Consultant, Heidenheim, Germany*
*[2] Siemens AG, CT SE 2, Munich, Germany*
*voelter@acm.org, iris.groher.ext@siemens.com*

## Abstract

*Software product line engineering aims to reduce development time, effort, cost, and complexity by taking advantage of the commonality within a portfolio of similar products. The effectiveness of a software product line approach directly depends on how well feature variability within the portfolio is implemented and managed throughout the development lifecycle, from early analysis through maintenance and evolution. Using DSLs and AO to implement product lines can yield significant advantages, since the variabilities can be implemented in higher level, less detailed models. This paper illustrates how variabilities can be implemented in model-to-model transformations and code generators. The backbone of the presented approach is to use aspect-oriented techniques for transformations and generators. These techniques are important ingredients for the model-driven product line engineering approach presented in [SPLC Paper].*

## 1 Introduction and Motivation

Most high-tech companies provide products for a specific market; thus the products have many things in common. An increasing number of these companies realize that product line development [1,2] fosters reuse at all stages of the lifecycle, shortens development time and helps staying competitive.

The effectiveness of a software product line approach directly depends on how well feature variability within the portfolio is managed from early analysis to implementation and through maintenance and evolution. Commonalities, as well as the flexibility to adapt to different product requirements are captured in core assets. Those reusable assets are created during domain engineering. During application engineering, products are either automatically or manually assembled, using the assets created during the domain engineering process and completed with product-specific artifacts. Products usually differ by the set of features they include in order to fulfill customer requirements. A feature is an increment in functionality provided by one or more members of a product line [3].

Variability management is the activity concerned with identifying, designing, implementing, and tracing flexibility in software product lines (SPLs). Variability of features often has widespread impact on multiple artifacts in multiple lifecycle stages, making it a predominant engineering challenge in software product line engineering (SPLE).

In traditional SPLE approaches, variability is mainly handled using either mechanisms provided by the implementation language, such as patterns, frameworks, polymorphism, reflection, and pre-compilers or using configuration and build tools to set compile time variables and select variants of assets. The approach described in this paper facilitates variability implementation, management, and tracing from architectural modeling to implementation of product lines by integrating both model-driven (MDSD) and aspect-oriented software development (AOSD). Here is a definition of what we call model-driven, aspect-oriented product line engineering:

> *MDD-AO-PLE uses models to describe product lines. Variants are defined on model-level. Transformations generate running applications. AO techniques are used to help define the variants in the models as well as in the transformers and generators.*

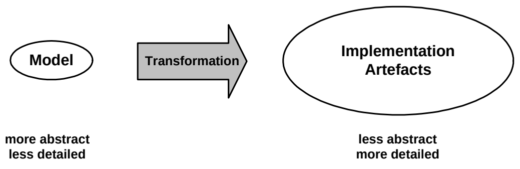

*Figure 1: Mapping abstract models to detailed implementations*

The core idea is to express variability in models and generators, since, as a consequence of the higher abstraction level in models (Figure 1), the number of variation points is lower (Figure 2).

For companies that are already building product lines, MDSD and AOSD can further increase productivity because:

- Variability can be described more concisely since in addition to the traditional mechanisms, variability is also described on model level.
- The mapping from problem to solution domain can be formally described and automated using model-

to-model transformations (Figure 3).

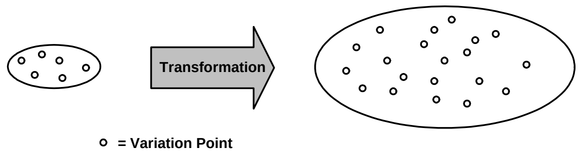

*Figure 2: Variation Point Mapping in PDD-PLE*

- Aspect-oriented techniques enable the explicit expression and modularization of crosscutting variability on model, code, and generator level.
- Fine grained traceability is supported since tracing is done on model element level rather than on the level of code artifacts.

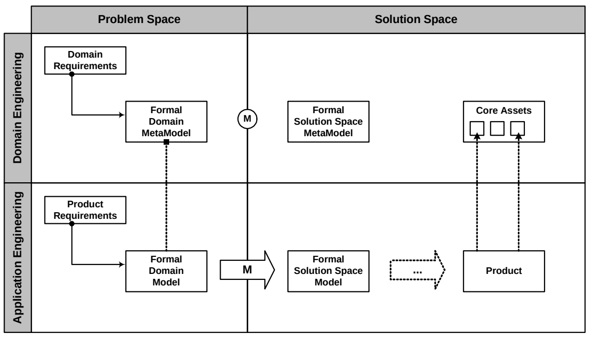

*Figure 3: The various models in MDD-AO-PLE*

The rest of the paper is organized as follows: The rest of section 1 introduces the main concepts of model-driven aspect-oriented product line engineering and introduces the case study as well as the tool environment we use. Section 2 is the main section of the paper and introduces transformation and generator aspects, and how they are coupled with a configuration model. Section 3 looks at related work, while section 4 summarizes the paper and provides and outlook on future work.

## 1.1 Concepts and Building Blocks

This paper explores an approach that integrates model-driven and aspect-oriented techniques in order to facilitate variability implementation, management and tracing in SPLE.

The general approach we are going to propose is as follows:

- Express as many artifacts as possible using models as this allows for processing these artifacts using model transformations.
- Mappings from problem to solution domain are implemented as model-to-model (M2M) transformations. This enables to formally describe mappings and automate their execution.
- Variable parts of the resulting system are either assembled from pre-build assets generated from models or implemented via interpreters. This is more efficient and less error-prone than manual coding in a third generation language (3GL).
- Aspect-oriented modeling (AOM) [8,11] is used to implement variability in models. This supports the selective adaptation of models. Details on this can be found in [IrisPaper]
- AO techniques are used to define variants of transformations and code generators.

A more detailed description of this overall approach is presented in [SPLC paper]. This paper provides details on the last of these points. Specifically, it showcases the tools we use for implementing these techniques. Techniques for building variants of models are described in [Iris Paper].

This paper uses a case study to illustrate the concepts.

## 1.2 Introduction to Case Study: Home Automation

The case study to illustrate our approach is a home automation system (see also [1]), called *Smart Home*. In homes you will find a wide range of electrical and electronic devices such as lights, thermostats, electric blinds, fire and smoke detection sensors, white goods such as washing machines, as well as entertainment equipment. Smart Home connects those devices and enables inhabitants to monitor and control them from a common UI. The home network also allows the devices to coordinate their behavior in order to fulfill complex tasks without human intervention.

Sensors are devices that measure physical properties of the environment and make them available to Smart Home. Controllers activate devices whose state can be monitored and changed. All installed devices are part of the Smart Home network. The status of devices can either be changed by inhabitants via the UI or by the system using predefined policies. Policies let the system act autonomously in case of certain events. For example in case of smoke detection windows get closed and the fire brigade is called. Varying types of houses, different customer demands, the need for short time-to-market and saving of costs drive the need for a Smart Home product line and are the main causes of variability.

## 1.3 Introduction to the Tooling

A central goal of our work is to build usable tooling for the concepts we introduce. It is important that the tooling is usable and available as widely as possible. Hence we're building the tooling on top of widely used open source tools, naming Eclipse (including the Eclipse Modeling Framework, EMF [ref]) and openArchitectureWare [ref].

In this paper we will be talking specifically about three parts of openArchitectureWare. Let's introduce them briefly:

- Workflow files are XML files that describe the steps that need to be executed in a generator run. Each of these steps is specified with what we call a workflow component. A typical oAW workflow consist of loading one or more models, checking constraints on them, transforming them into another model and then generating code from them.
- Code generation in oAW is done with a language called Xpand. It is an object-oriented template language. An Xpand file consists of a number of templates, each of them declared by a *DEFINE name FOR metaclass* clause.
- Model-to-Model transformation is done with a language called Xtend. It is a textual and (more or less) functional language for querying and navigating existing models as well as building new models. The expression sub-language is a simplified version of OCL.

## 2 Building Variants of Transformations and Generators

The following section illustrates various ways of building variants of transformers and generators. While the mechanisms are different in the way they change the actual behavior of the transformation or generator, they have one thing in common: The behavior change they implement is only applied to the system if a certain feature is selected in our configuration feature model. This is a form of orthogonal variability [ref]. A central feature model (Figure 4) represents all the configurative variability for our family of transformers/generators.

In our tooling, this feature model (and the corresponding configuration models, Figure 5) is implemented using pure::variants [ref] (other tools could be used – the dependency is well isolated).

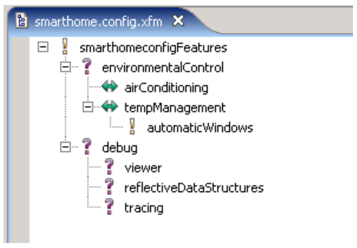
```
smarthome.config.xfm
smarthomeconfigFeatures
  environmentalControl
    airConditioning
    tempManagement
      automaticWindows
  debug
    viewer
    reflectiveDataStructures
    tracing
```

*Figure 4: Part of the Feature Model*

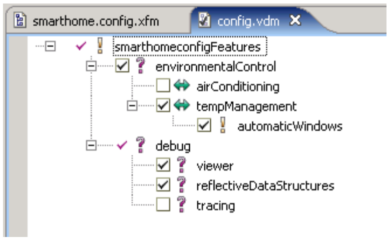
```
smarthome.config.xfm   config.vdm
✓ smarthomeconfigFeatures
  [✓] environmentalControl
    [ ] airConditioning
    [✓] tempManagement
      [✓] automaticWindows
  ✓ debug
    [✓] viewer
    [✓] reflectiveDataStructures
    [ ] tracing
```

*Figure 5: A specific configuration*

### 2.1 Variants of M2M Transformations

In our case study, the solution space model is built from component instances connected by connectors. A model transformtion creates these models from a problem space model that contains buildings and their SmartHome equipment. Figure 6 shows the process for an example building. Our component framework supports interceptors. It is possible to configure a set of interceptors into a set of component instances. Hence, whenever an operation is invoked on a component instance, the interceptor is notified and can execute *before* and *after* behavior.

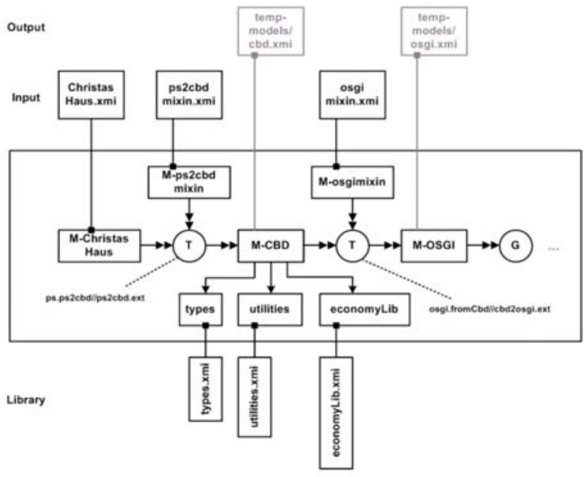

*Figure 6: Example Transformation Process*

So, in order to add logging (or anything else that can be handled via an interceptor) to the system we need to make sure a suitable interceptor is configured into the respective component instances. The way we do this is to write a *transformation aspect* that advices the problem space to solution space transformation accordingly. The transformation aspect is only applied to the transformation workflow if the respective feature is selected in the configuration model. Figure 7 shows a thumbnail of the approach.

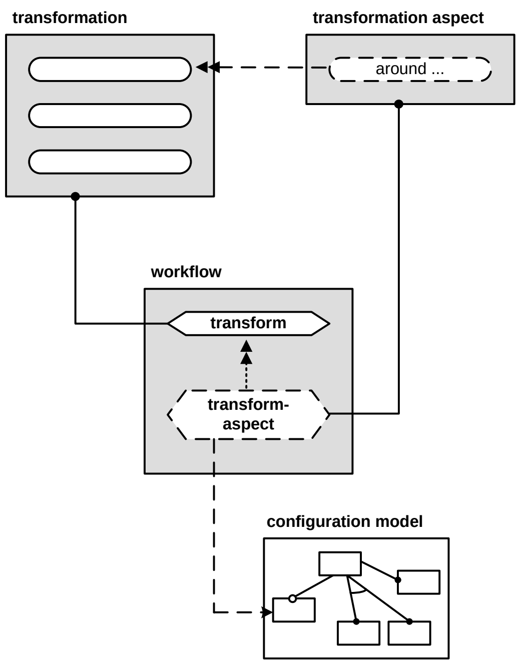

*Figure 7: Implemting the Logging Feature*

**Implementing the transformation aspect**

The aspect that actually modifies the transformation is shown in the following piece of code. It is implemented in oAW's Xtend language.

```
extension ps2cbd;

around ps2cbd::transformPs2Cbd( Building building ):
 let s = ctx.proceed(): (
   building.createBuildingConfiguration().
     deployedInterceptors.addAll(
        { utilitiesib().interceptors.findByName("TracingInterceptor") }
     ) -> s
  );
```

In this aspect, we advice the *ps2cbd::transformPs2Cbd* function which is the "main method" of the problem space to solution space transformation used in the system. Inside the advice, we execute the original definition (*ctx.proceed()*) and then we add the *TracingInterceptor* to the list of deployed interceptors of the top level configuration. Interceptors are loaded from a library of reusable components. A configuration is a container for a set of component instances; instances inherit the interceptors configured in their owning configuration.

**Connecting the aspect with the configuration**

Remember that we only want to have these interceptors in the system iff the feature *logging* is selected in the global configuration model. This dependency is expressed in the workflow.

Somewhere in that workflow, an *XtendComponent* is used to execute the original problem to solution space transformation:

```
<component id="xtendComponent.ps2cbd"
            class="oaw.xtend.XtendComponent">
 ...
</component>
```

We now need to make sure that this *XtendComponent* is aware of the aspect we want to add to the transformation in order to add the interceptor. However, we don't want to modify the declaration of the actual workflow component as shown above, since that would mean an invasive change to an existing workflow file. For reasons of modularity, this is something we need to avoid. Consequently, the oAW workflow language also supports aspects. Here is the workflow code we need to write:

```
<feature exists="logging">
  <component adviceTarget="xtendComponent.ps2cbd"
              class="oaw.xtend.XtendAdvice">
     <extensionAdvices value="logging"/>
  </component>
</feature>
```

The *XtendAdvice* component is used to add additional sub-elements to the component referenced by the *adviceTarget* attribute (which references the *XtendComponent* declared above). However, that component is only seen by the workflow engine if the feature *logging* exists. This is expressed by the surrounding *<feature…>* tag.

## 2.2 Variants of Code Generators

In this section we will look at building variants of code generators. oAW uses a template-based code generator, which is why a code generator is not the same as a model-to-model transformation (we do *not* instantiate the AST of the target language).

Let us look at another example from the SmartHome case study. In order to debug and control the demonstrator, we can run a GUI with the generated application. The GUI itself is not generated. It is part of the platform and accesses the system using reflection. In order for it to be able to do this, certain parts of the system need to include a specific reflection layer that is used by that GUI. Specifically, if you want to be able to inspect component instance states, the following two things need to be done:

- The data structures representing the state need to be "reflective"
- Upon system startup, the state objects of each instance need to be registered with the GUI

Of course, since this functionality is for debugging purposes only, it is optional – i.e. depending on whether a certain feature is selected. Figure 8 shows the thumbnail of the solution.

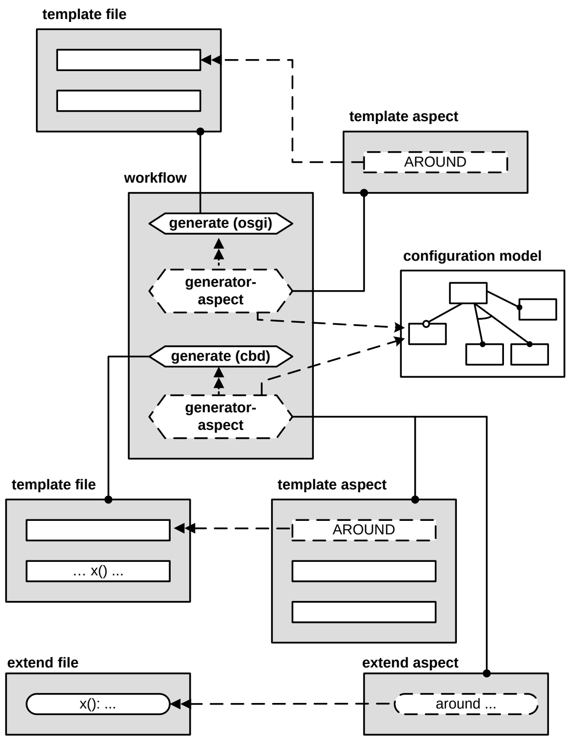

*Figure 8: Implementing the features necessary for the debug GUI*

In the following, we will only show the code generator aspect that is used to add the reflection layer to the state data structures.

The code generator for the data structures contains the following templates. *typeClass* generates a Java class that represents the state data structure (basically a bean with getters and setters). That template in turn calls the *imports* and *body* templates. Those will be the templates that will be advices by the aspect shown below.

```
«DEFINE typeClass FOR ComplexType»
  «FILE fileName()»
    package «implClassPackage()»;
    «EXPAND imports»
    public class ... {
      «EXPAND body»
    }
  «ENDFILE»
«ENDDEFINE»

«DEFINE imports FOR ComplexType»
«ENDDEFINE»

«DEFINE body FOR ComplexType»
   ...
«ENDDEFINE»
```

The following piece of Xpand code is the template aspect that adds the reflection layer to the generated data structures. Note how the *AROUND* declarations reference existing *DEFINE*s in order to advice them. *targetDef.proceed()* calls the original template.

```
«AROUND data::api::data::body FOR ComplexType»
  «targetDef.proceed()»
  «EXPAND reflectionImplementation»
«ENDAROUND»

«AROUND data::api::data::imports FOR ComplexType»
 «targetDef.proceed()»
 import smarthome.common.platform.MemberMeta;
 import smarthome.common.platform.ComplexTypeMeta;
«ENDAROUND»

«DEFINE reflectionImplementation FOR ComplexType»
 private transient ComplexTypeMeta __meta = null;
 public ComplexTypeMeta __metaObject() {
   ...
 }
 public void __metaSet( MemberMeta member, Object value ) {
   ...
 }
 public Object __metaGet( MemberMeta member ) {
   ...
 }
«ENDDEFINE»
```

Of course, to make this work as desired, we have to couple the aspect to the configuration model. We do this by modifying the workflow in the same way as in case of the M2M aspects:

- We add a *GeneratorAadvice* component (as opposed to an XtendAdvice, since we now want to advice a code generator, and not a model-to-model transformation). It specifies the original *Generator* as its advice target and makes the Xpand file with the AROUND templates known.
- We encapsulate this *GeneratorAdvice* with a *<feature…>* tag to make it depend on a certain feature in the configuration model.

## 2.3 More features

This section introduces a couple of additional features that don't deserve their own section.

**Querying the feature model directly**

In addition to the tooling introduced above, we can also access the configuration model directly from within transformations or code generation templates. For example, the following piece of transformation code optionally adds burglar detection facilities to our building. The same function can be called from inside a template (typically, as part of an *IF* statement).

```
create System transformPs2Cbd( Building building ):
 ...
 hasFeature("burglarAlarm") ? ( handleBurglarAlarm() -> this) : this;


handleBurglarAlarm( System this ):
 let conf = createBurglarConfig(): (
   configurations.add( conf ) ->
   ...
   conf.connectors.add( connectSimToPanel( createSimulatorInstance(),
                              createControlPanelInstance() ) ) ->
   hasFeature( "siren" ) ? conf.addAlarmDevice("AlarmSiren") : null ->
   hasFeature( "bell" ) ? conf.addAlarmDevice("AlarmBell") : null ->
   hasFeature( "light" ) ? conf.addAlarmDevice("AlarmLight") : null
);
```

In principle, the *hasFeature()*-based approach shown here is similar to the AOP-based approach introduced in the previous section. You can handle each variability with both facilities. But just as in regular programming, there are tradeoffs a developer has to consider:

- The conditional *hasFeature()* is simpler, but requires invasive changes to existing transformations (which you might not be able to do because they are bought as part of a third party catridge). Especially in cases where you have to query for the same feature in many locations, this creates a maintenance nightmare.
- The AO-based approach is a bit more complex (write the aspect, write the workflow aspect, tie it to the feature model) but supports non-invasive changes. Also, if a given feature requires several advises targeting different locations in existing assets, all of these advices can be bundled in the same Xtend or Xpand file, thereby enhancing feature modularity significantly.

**Feature Attributes**

It is also possible to address properties or attributes of features. For example, you might want to be able to configure the volume of the siren in the configuration model. The transformation would read this value from the configuration model and parametrize the siren component instance accordingly. Here's the code:

```
handleBurglarAlarm( System this ):
 ...
 isFeatureSelected( "siren" ) ? (
    let siren = conf.addAlarmDevice("AlarmSiren"):
      siren.configParamValues.add( siren.createParamForLevel() )
    ) : null ->
   ...
);

private create ConfigParameterValue
      createParamForLevel( ComponentInstance instance ):
  setName( "level" ) ->
  setValue((String)getFeatureAttributeValue( "siren", "level" ));
```

**Quantification in the Aspects**

An important characteristic of AO is that a given aspect should be able to not just advice one specific join point in the base system, but rather query the base system and advice a set of matching join points. Although we think this feature is not very important for building variants of generators (on the meta level, there's less crosscutting), oAW's AO facilities for Xtend and Xpand support polymorphic matching as well as wildcards in the name of the adviced entity.

## 3 Related Work

Let us first look at the related work developed and published by us. The SPLC paper [SPLC] explains the general idea of model-driven aspect oriented product line engineering, and how the case study illustrates the approach overall. While the paper you're currently reading looks at building variants of generators, the paper [Iris Paper] looks at the other important ingredient: building variants of models. These two techniques together form the backbone of the MD-AO-PLE.

TODO: Other People's work

## 4 Summary and Future Work

In this paper we have presented an approach to build families of generators. The main tools for implementing the respective variability are

- Isolation of the variant-specific code (transformation or template) into a separate file, a transformation or generator aspect aspect
- Contribute that aspect to an existing workflow file without changing the original workflow file using *XtendAdvice* and *GeneratorAdvice* components.
- Implement orthogonal variability of aspects and workflows by making the deployment of the aspects depend on the presence of certain features in a configuration model.

Our next steps will be concerned with implementing better tooling for the features we've introduced in this paper. The tooling with make working with feature-dependencies more effective, for example by

- Finding all the workflow components that depend on a given feature

- Find the workflow component that is addressed by an *adviceTarget* attribute of an advice component

Many of the tooling improvements will also concern the variability management in models, as described in [Iris Paper].

## 5 Acknowledgments

This work is supported by AMPLE Grant IST-033710. References

## 6 Refernces

[1] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering Foundations, Principles, and Techniques.* Berlin: Springer, 2005.

[2] P. Clements, and L. M. Northrop, *Software Product Lines: Practices and Patterns:* Addison Wesley, 2001.

[3] P. Zave, "FAQ Sheet on Feature Interaction": http://www.research.att.com/~pamela/faq.html

[4] T. Stahl, and M. Voelter, *Model-Driven Software Development:* Wiley & Sons, 2006.

[5] R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development.* Amsterdam: Addison-Wesley Longman, 2004.

[6] AOSD website, http://www.aosd.net

[7] M. Voelter, "Patterns for Handling Cross-cutting Concerns in Model-Driven Software Development", In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP).* Irsee, Germany, July, 2005.

[8] Aspect-Oriented Modelling Workshops website, http://www.aspect-modeling.org/

[9] First Workshop on Models and Aspects – Handling Crosscutting Concerns in MDSD, ECOOP, Glasgow, UK, July, 2005.

[10] Second Workshop on Models and Aspects – Handling Crosscutting Concerns in MDSD, ECOOP, Nantes, France, July, 2006.

[11] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design. The Theme Approach.* Amsterdam: Addison-Wesley Longman, 2005.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with ASPECTJ," *Communications of the ACM*, vol. 44, pp. 59 - 65, October 2001.

[13] I. Aracic, V. Gasiunas, K. Ostermann, and M. Mezini, "An Overview of CaesarJ" in *Transactions on AO Software Development I.* vol. 3880/2006 Berlin/Heidelberg Springer, 2006, pp. 135-173.

[14] OMG Query/Views/Transformations (QVT) specification, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[15] K. Czarnecki and U. W. Eisenecker, *Generative Programming. Methods, Tools, and Applications.* Amsterdam: Addison-Wesley Longman, 2000.

[16] P. Maeder, M. Riebisch, and I. Philippow, "Traceability for Managing Evolutionary Change - A Roadmap", In *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE).* Los Angeles, USA, July, 2006.

[17] K. Mohan and B. Ramesh, "Managing Variability with Traceability in Product and Service Families", In *Proceedings of the 35th Hawaii International Conference on System Sciences (HICCS).* Hawaii, January, 2002.

[18] M. Voelter, "A Collection of Patterns for Program Generation", In *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP).* Irsee, Germany, July, 2003.

[19] OSGi Alliance website, http://osgi.org

[20] M. Pinto, L. Fuentes, and J. M. Troya, "A Component and Aspect Dynamic Platform", *The Computer Journal*, vol. 48(4), pp. 401-420, 2005.

[21] Eclipse Foundation website, http://eclipse.org

[22] Eclipse Modeling Framework (EMF) website, http://eclipse.org/emf

[23] openArchitectureWare (oAW) website, http://www.eclipse.org/gmt/oaw/

[24] XFeature Feature Modelling Tool website, http://www.pnp-software.com/XFeature/

[25] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-in for Eclipse", In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, OOPSLA, Vancouver, British Columbia, Canada, Pages 67 - 72, ACM Press, 2004.

[26] pure::variants Variant Management Tool website, http://www.pure-systems.com/3.0.html

[27] Eclipse Graphical Modeling Framework (GMF) website, http://eclipse.org/gmf

[28] F. Jouault and J. Bézivin, "On the Specification of Textual Syntaxes for Models", In *Proceedings of the Eclipse Summit Europe (Eclipse Modeling Symposium)*, Esslingen, Germany, October, 2006.

[29] ATL Model Transformation Language website, http://www.eclipse.org/m2m/atl/

[30] Eclipse M2M project website, http://www.eclipse.org/m2m/

[31] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants", In *Proceedings of the 4th Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September, 2005, pp. 422 - 437, Springer, 2005.

[32] M. Didonet del Fabro, J. Bézivin and P. Valduriez, "Weaving Models with the Eclipse AMW plugin", In *Proceedings of the Eclipse Summit Europe (Eclipse Modeling Symposium)*, Esslingen, Germany, October, 2006.

[33] I. Groher and M. Voelter, "XWeave – Models and Aspects in Concert", In *Proceedings of the 10th Workshop on AO Modeling*, Vancouver, Canada, March, 2007.

[34] M. Mezini and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects", In *Proceedings of the 12th International Symposium on Foundations of Software Engineering (FSE)*, Newport Beach, CA, USA, 2004, pp. 127-136.

[35] S. Apel, T. Leich, and G. Saake, "Aspectual Mixin Layers: Aspects and Features in Concert", In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, 2006, pp. 122-131.

[36] N. Loughran and A. Rashid, "Framed Aspects: Supporting Variability and Configurability for AOP", *In Proceedings of the 8th International Conference on Software Reuse*

*(ICSR)*, Madrid, Spain, 2004.
[37] DOORS Requirements Management Tool website, http://www.telelogic.com/products/doors/
[38] N. Ubayashi, S. Sano, Y. Maeno, S. Murakami, and T. Tamai, “Model Evolution with Aspect-Oriented Mechanisms”, In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE)*, Lisbon, Portugal, 2005, pp. 187-194.
[39] J. Liu, R. Lutz, and H. Rajan, “The Role of Aspects in Modeling Product Line Variabilities”, In *Proceedings of the 1st Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, GPCE, Portland, Oregon, October, 2006.
[40] K. Czarnecki, M. Antkiewicz, C.H.P. Kim, S. Lau, and K. Pietroszek, “Model-Driven Software Product Lines”, Poster Session, OOPSLA, San Diego, USA, October, 2005.
[41] Model-Driven Architecture (MDA) website, http://www.omg.org/mda/