Handling Variability in Model Transformations and Generators

Markus Voelter[1], Iris Groher[2]
[1] Independent Consultant, Goeppingen, 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 variability can be implemented on a higher level of abstraction, in less detailed models. This paper illustrates how variability can be implemented in model-to-model transformations and code generators using aspect-oriented techniques. These techniques are important ingredients for the aspect-oriented model-driven product line engineering approach presented in [13].

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 [2].

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, [3]) and aspect-oriented software development (AOSD, [4]). Here is a definition of what we call aspect-oriented model-driven 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.

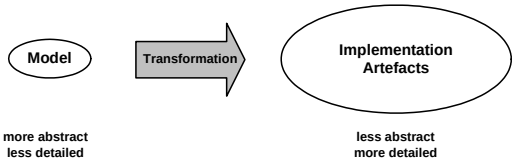

Figure 1: Mapping abstract models to detailed implementations

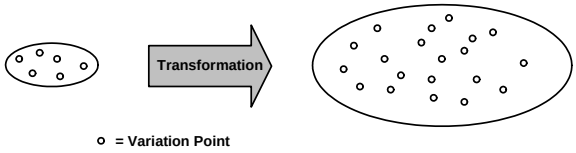

Figure 2: Variation Point Mapping in Model-Driven PLE

The core idea is to express variability in models and generators, since, due to 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.
- 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 illustrates the various models and meta models used in our approach. In the problem space we use a problem domain specific meta model (and conseuqently, a DSL) to describe domain concepts. In application engineering this meta model is instantiated to describe specific applications. This model is then transformed into a solution space model, which is an instance of a solution space meta model defined during domain engineering. Finally, the solution space model is transformed into the product (i.e. typically source code). The product uses pre-built assets also created during domain engineering.

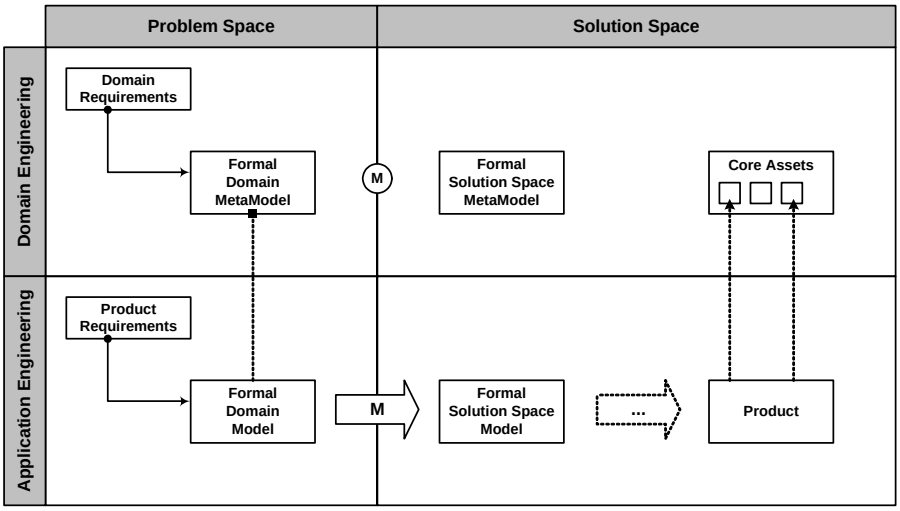

Figure 3: The various models in AO-MD PLE

The rest of the paper is organized as follows: The remainder of Section 1 introduces the main concepts of aspect-oriented model-driven product line engineering and demonstrates the case study as well as the tool environment we use. Section 2 illustrates transformation- and generator-level aspects and how they are linked to a configuration model. Section 3 looks at related work, while Section 4 summarizes the paper and provides an 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) [5, 11] is used to implement variability in models. This supports the selective adaptation of models. Details on this can be found in [14]
- AO techniques are used to define variants of transformations and code generators.

A more detailed description of the overall approach is presented in [13], while this paper focuses on the last point. Specifically, it showcases the tools we use for implementing variability in transformations and generators. Techniques for building variants of models are described in [14].

The concepts are illustrated with a case study of a home automation system product line.

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 new 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 commonly used open source tools, namely Eclipse (including the Eclipse Modeling Framework, EMF [6]) and openArchitectureWare (oAW) [7].

In this paper we are focusing on three parts of openArchitectureWare:

- 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 consists of loading one or more models, checking constraints on them, transforming them into other models and then generating code from them.
- Code generation in oAW is done using a language called Xpand [7]. 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 transformations are done using a language called Xtend [7]. 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

This 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 important 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 [1]. A central variability model (Figure 4) represents all the configurative variability for our "tool product line".

In our tooling, the feature model (and a specific selection of features, Figure 5) is implemented using pure::variants [10] (other tools such as [8, 25, 11] could easily be used – the variant management tool dependency is well isolated).

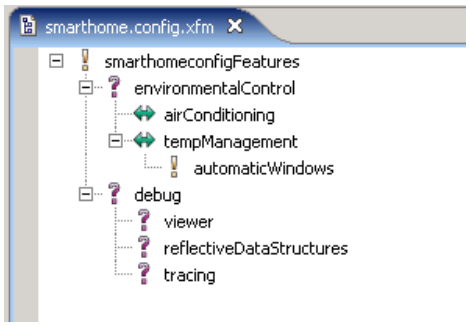

Figure 4: Part of the Feature Model

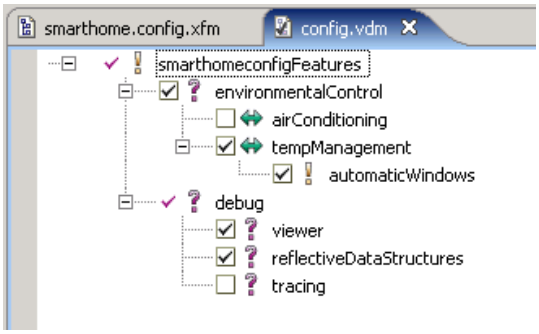

Figure 5: A specific configuration

2.1 Variants of M2M Transformations

In our case study, the solution space model is built from component instances linked by connectors. A model transformtion creates these models from a problem space model that contains buildings and their respective Smart Home 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.

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.

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.

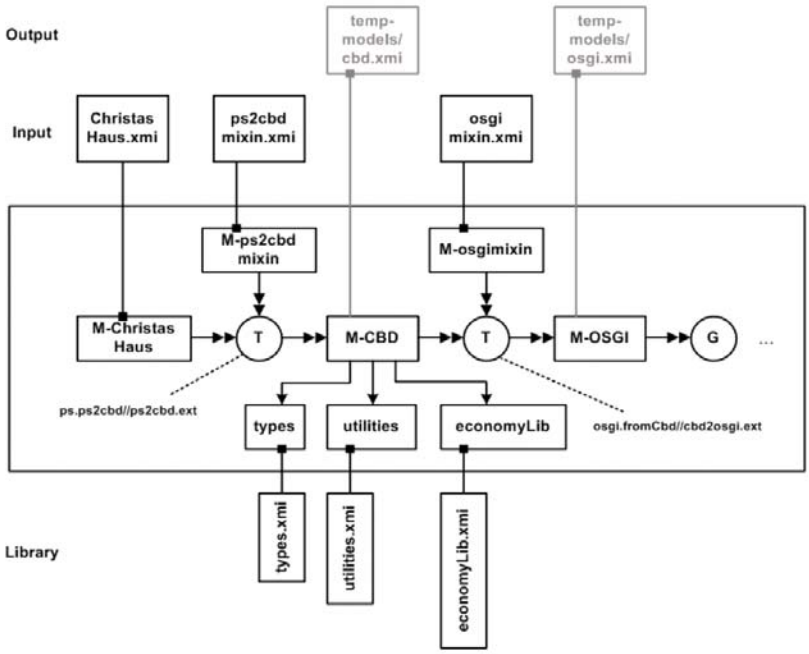

Figure 6: Example Transformation Process

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 optional interceptor.

However, we don't want to modify the declaration of the actual workflow component as shown above, since that would lead to 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 executed by the workflow engine if the feature *logging* is selected in the configuration model. This is expressed by the surrounding <*feature…*> tag.

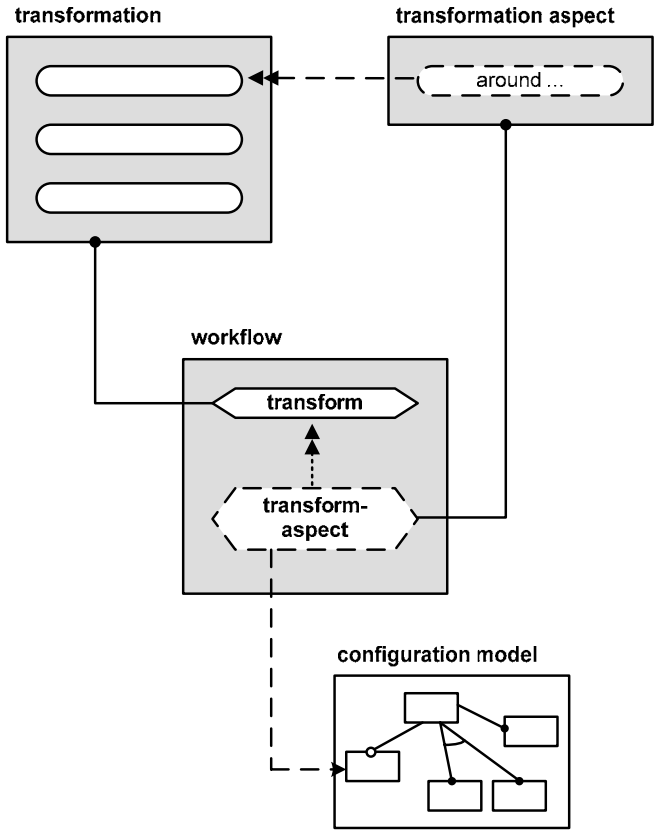

Figure 7: Implementing the Logging Feature

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 transformator (we do *not* instantiate the AST of the target language).

Let us look at another example from the Smart Home 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. it depends on whether a certain feature is selected. Figure 8 shows the thumbnail of the solution.

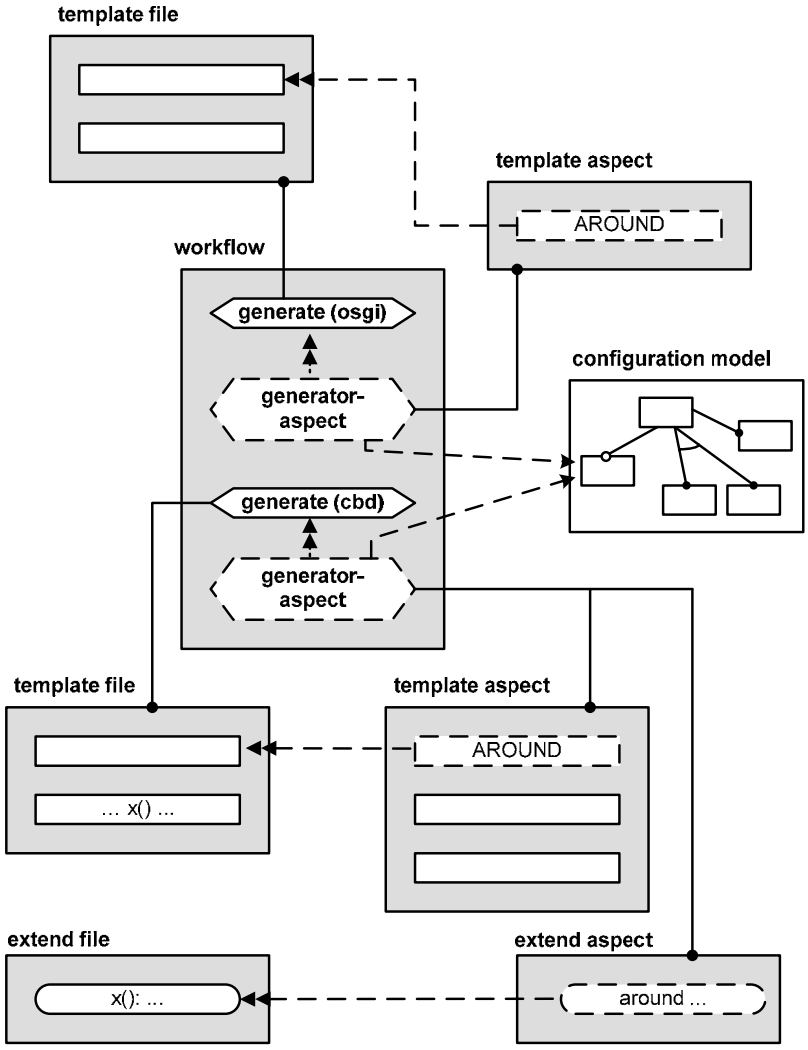

Figure 8: Implementing the features necessary for the debug GUI

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 adviced 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 the case of the M2M transformation aspects:

- We add a *GeneratorAdvice* 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 dependent on a certain feature in the configuration model.

2.3 More features

This section introduces a couple of additional features that don't need a separate subchapter.

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 has the same effect as the AOP-based approach introduced in the previous section. Each variability can be handled 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 might be bought as part of a third party cartridge). Especially in cases where you have to query for the same feature in many locations, this creates a maintenance nightmare. The conditional is scattered over many places in the transformations.
- 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 advices 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 parameterize 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 [13] explains the general idea of aspect-oriented model-driven product line engineering, and how the case study implements the overall approach. While this paper looks at building variants of generators, another paper [14] looks at the other important ingredient: building variants of models. These two techniques together form the backbone of AO-MD-PLE. The AMPLE research project [12], in the context of which this research has been conducted, is slated for more research in this area and is therefore worth following.

The transformation engine of ArcStyler, CARAT [18], allows the specialization of cartridges. It allows to override "generator code" specified in a super cartridge. While this kind of specialization is also possible with our approach, our approach is more generic because it also support quantification – the ability to select a set of join points using a pointcut to change behavior in several places at once.

There are existing approaches that also use AO together with generative techniques. However, in contrast to our approach, they generate AOP code and hence implement the variants on the level of the generated artifact. The limitation of that approach is that it only works for target languages that have aspect support, whereas our approach does not require this. [15] uses "generative aspects" to enhance COTS generators. They generate AspectJ code from the same model from which the original generator had built the components; the generated aspects customize the code generated by the COTS compiler. In [16], aspects and generative approaches are also combined. Here, the generated aspects are used to specialize potentially complex framework classes. [17] uses generated aspects to customize reusable components.

4 Summary and Future Work

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

- Isolation of the variant-specific code (transformation or template) into a separate file, a transformation or generator aspect.
- Contribute that aspect to an existing workflow file without changing the original workflow file using *XtendAdvice* and *GeneratorAdvice* components.
- Implement orthogonal and configurative 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 improving the tooling we've introduced in this paper. The tooling will effectively improve working with feature-dependencies, for example by finding all the workflow components that depend on a given feature or finding the workflow component that is addressed by an *adviceTarget* attribute of an advice component.

Many of the tool improvements will also concern the variability management in models, as described in [14].

5 Acknowledgments

This work is supported by AMPLE Grant IST-033710.

6 References

[1] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering Foundations, Principles, and Techniques.* Berlin: Springer, 2005.
[2] P. Zave, *FAQ Sheet on Feature Interaction*: http://www.research.att.com/~pamela/faq.html
[3] T. Stahl, and M. Voelter, *Model-Driven Software Development:* Wiley & Sons, 2006.
[4] AOSD website, http://www.aosd.net
[5] Aspect-Oriented Modelling Workshops website, http://www.aspect-modeling.org/
[6] Eclipse Modeling Framework (EMF) website, http://eclipse.org/emf
[7] openArchitectureWare (oAW) website, http://www.eclipse.org/gmt/oaw/
[8] XFeature Feature Modelling Tool website, http://www.pnp-software.com/XFeature/
[9] 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, Canada, Pages 67 - 72, ACM Press, 2004.
[10] pure::variants Variant Management Tool website, http://www.pure-systems.com/3.0.html
[11] BigLever, Gears Variant Management Tool website, http://www.biglever.com
[12] AMPLE Research Project, http://www.ample-project.net/
[13] M. Voelter, and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Developent", *In Proceedings of the 11th International Software Product Line Conference (SPLC)*, Kyoto, Japan, 2007.
[14] I. Groher, and M. Voelter, "Expressing Feature-Based Variability in Structural Models", *Submitted for publication*, 2007.
[15] Cody Henthorne , Eli Tilevich, "Code Generation on Steroids: Enhancing COTS Code Generators via Generative Aspects", *In Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS '07)*, Minneapolis, USA, May 2007
[16] A. L. Santos, A. Lopes, K. Koskimies, "Framework Specialization Aspects", *In Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, Vancouver, Canada, March, 2007
[17] X. Liu , Y. Feng , J. Kerridge, "Achieving Dependable Component-Based Systems Through Generative Aspect Oriented Component Adaptation",*In Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Chicago, USA, September 2006
[18] iO Software, ArcStyler CARAT Guide for ArcStyler Version 5, http://www.io-software.de/support/doc/doc/Carat_Guide.pdf