Using Transformation-Aspects in Model-Driven Software Product Lines

Hugo Arboleda, Rubby Casallas, Jean-Claude Royer

Using Transformation-Aspects in Model-Driven Software Product Lines*

Hugo Arboleda[1,2], Rubby Casallas[2], and Jean-Claude Royer[1]

[1] École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes Cedex 3, France
{hugo.arboleda,jean-claude.royer}@emn.fr
[2] University of Los Andes, Cra. 1 N◦ 18A 10, Bogotá, Colombia
rcasalla@uniandes.edu.co

Abstract. Model-Driven Software Product Lines (MD-SPL) are configured by using configuration models and Problem Space metamodels that capture product line scope. Products are derived by means of successive model transformations, starting from problem space models and based on the configuration models. Fine-variations of MD-SPLs correspond to characteristics that affect particular elements of models involved in the model transformations. In this paper, we present an approach to create MD-SPL including fine-variations. We configure products creating fine-feature configurations. Then, based on such configurations, we create MD-SPLs using principles of Aspects Oriented Development. Thus, our approach allows to derive products including fine-grained details of configuration.

Key words: Model Driven Development, Software Product Lines, Variability Management, Product Derivation, Fine-Grained Variability.

1 Introduction

Aspect-oriented programing (AOP) [1–3] improves software development by providing constructs for the encapsulation of crosscutting concerns. Aspects encapsulate crosscutting concerns and are subsequently composed with other software artifacts using composition mechanisms. A join point model captures the set of possible composition points for a specific aspect. Aspects are automatically composed with the rest of the system by an aspect weaver. Asymmetric AOP approaches such as AspectJ [1] provide constructs for the encapsulation of crosscutting concerns that are woven to some (non-AOP) base system. A recent work [4, 5] has shown that AOP is a valuable complement for Model Driven Development (MDD) [6] regarding the automatic generation of software products. First, model transformation programs can be modularly adapted [5, 7], based on approaches as the one introduced by AspectJ. Second, Aspect Oriented Modeling (AOM) techniques [8–11] provide support to implement variability at model

* This work has been supported by the European Commission STREP Project AMPLE IST-033710 and the "Instituto Colombiano para el Desarrollo de la Ciencia y la Tecnología - Francisco José de Caldas (COLCIENCIAS)"

level. This means that different models can be adapted applying a weaving that crosscuts their concerns.

MDD has proved benefits regarding the derivation of Software Product Lines (SPLs). In Model Driven Software Product Line approaches (MD-SPL) [12–14], product families are derived by means of chains of model transformations, starting from an initial domain model and based on configuration models, until to get a final solution model. Current Aspect Oriented (AO-)MD-SPL approaches, as those introduced by Voelter et al. [7], deal with the problem of creating products including variations that affect the whole product. We call these, *coarse-grained variations*. For example, a coarse-grained variation is a property as internationalization (English or German), or in a SPL of Smart Home systems, a coarse-grained variation expresses that a house has automatic lights. This implies that for a particular Smart Home system, "all" the instances of a light component have functionality of automatic lights.

In our work, it is important to distinguish *coarse-grained variations* from *fine-grained variations*. Fine-grained variations are characteristics that affect particular elements of models involved in model transformations. For example, for a particular Smart Home system, it is possible to select the instances of a light component that have functionality of automatic lights. In addition, it is possible that such instances manage different attributes to configure the intensity of the lights when they are turned on. Thus, it is possible to have an automatic light in the living room that is turned on at 19h00, and another automatic light in the main room that is turned on in the presence of an inhabitant. The expression of fine-grained variations and the use of it to subsequently derive products are important activities in the SPLs creation process. This is because customer requirements can be very specific, especially in the domain of software intensive systems where, *(i)* each component instance can manage variable functionality, and *(ii)* variability related to each component instance can imply expenses for customers.

In this paper, we improve the AO-MD-SPL approach presented by Voelter et al. [7], facilitating the creation of products including fine-grained variations. We express fine-grained variations and define its scope by using *constraint models*; and, we configure products including fine-grained variations by using *fine-feature configurations*. We also include a solution for the problem of deriving products based on fine-grained details of configuration. For this, we adapt the execution of model transformations applying *aspects* that include transformation logic to generate low-level configuration details. The aspects we create use a particular type of transformation rules that we have called *fine-transformation rules*. For instance, we adapt the execution of a model transformation that transforms elements conforming to a metaconcept `Room`, when some of such elements is affected by a fine-grained variation. The fine-grained variation can be that a `Room` element has to be transformed including an environmental control system as air conditioning.

We have implemented our approach using and extending the openArchitectureWare (oAW) [15] toolkit. We have chosen oAW, because this is a complete

MDD framework integrated into Eclipse that makes possible the reading and instantiating of models, checking them for constraint violations, and transforming them into other models or source code. Furthermore, oAW also provides support for AOM and AOSD in the context of MDD. Finally, the framework has been used successfully to create SPLs [5], and there is an active community of developers using and improving it [15].

The remainder of this paper is organized as follows. Section 2 introduces a practical example we use to illustrate our approach. Section 3 presents our proposal for expressing fine-grained variations and configure products including it. Section 4 explains our solution to create MD-SPLs including fine-grained variations. Section 5 presents conclusions.

2 Practical Example

In the remaining of this paper, we will use as example the case of a Smart Home Product Line which is part of the AMPLE Project [16]. The description of a house includes structural elements as floors, rooms, doors, windows and stairs. Houses also include electrical and electronic devices such as automatic lights, security devices as alarms, among others. These devices, and therefore their behavior, are optional features that product designers have to select and bind to other elements that already exist in the house. For instance, to derive a particular Smart Home system, the automatic lights can be bound to all rooms in the house (coarse-grained variation) and the security alarm system can be bound only to the main door entrance (fine-grained variation). Figure 1 presents part of a domain model of a specific Smart Home system used as example.

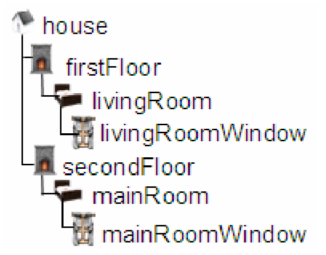

Fig. 1. Domain Model of a Smart Home.

The model is represented by a tree structure created using EMF [17]. This structure illustrates UML-type composition, for example, between the `house` and its `floors`. In the figure there are two floors, `firstFloor` and `secondFloor`. The `firstFloor` has a `livingRoom` and a `livingRoomWindow`; the `secondFloor` has a `mainRoom` and a `mainRoomWindow`. This model conforms to a domain metamodel that includes concepts of smart homes like `<<Building>>`, `<<Floor>>`, `<<Room>>` and `<<Window>>`.

3 Fine-Grained Variability

We distinguish between coarse-grained variations and fine-grained variations. The sentence *all the doors have fingerprint authentication system*, is an example of coarse-grained variation, and one of fine-grained variation is, *only the* *main door has a fingerprint authentication system*. To express and manage fine-variations, we introduce *constraint models*. To configure products including fine-variations, we introduce *fine-feature configurations*. We based the creation of constraint models and fine-feature configurations on feature modeling. In addition, we use feature modeling to express coarse-grained variations.

3.1 Feature Modeling

The basic mechanism we use to classify and describe configurable common and variable characteristics in the scope of a product line is feature models [18]. Specifically, we use the Czarnecki et al. [13] feature metamodel. In this feature metamodel, a `GroupFeature` expresses a choice over the set of `GroupedFeatures` in the group and its `cardinality` defines the restriction on the number of choices. For example, the `cardinality` [1..2][4..5] for a `GroupFeature` means that between one and two, and between four and five of its `GroupedFeature` can be chosen. A `GroupedFeature` does not have `cardinality`. A `SolitaryFeature` is a feature that is not grouped in a `FeatureGroup`. The `cardinality` of `SolitaryFeature` specifies the maximum number of times this feature can appear in the final feature configuration. Figure 2 presents a feature model including features of the SPL of Smart Home systems.

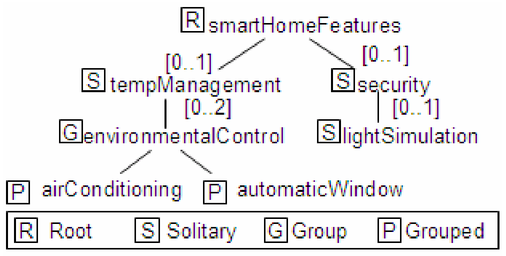

Fig. 2. Smart Home Feature Model.

Two `SolitaryFeatures`, `tempManager` and `security`, are at the root level. The `tempManager` feature has the `GroupFeature` `environmentalControl`, which has the `GroupedFeature` `airConditioning` and `automaticWindow`. The `security` feature has the `SolitaryFeature` `lightSimulation`.

3.2 Constraint Models

We call a *binding*, the materialization of a specific (fine-grained variation) choice for a product, expressed between elements of models and features. For example to express that, *the* `firstFloor` *has a central* `airConditioning` *that has to be turned on when the average temperature of the floor reaches* $22°C$, *and turned off when the temperature reaches* $18°C$.

Constraint models allow product line architects to restrict the bindings among elements and features that a product designer can establish. For example, to express that, *maximum three elements that conform to the metaconcept* `Window` *can be bound to the feature* `automaticWindow`.

A `constraint` $o = [M, F, P]$ is a tuple composed by a metaconcept M, a feature F, and a property P. A constraint o expresses the fact that model elements conforming to the metaconcept M can be bound to the feature F. We use the property P to define additional constraints that a model element has to satisfy to be bound to a feature. For example, the constraint `ConstraintAutoWindow`=(`Window`, `automaticWindow`, *cardinality-Temperature*) describes that, during the configuration of a specific Smart Home system, a product designer can bind elements conform to `Window`, for example the `mainRoomWindow`, to the `automaticWindow` feature. The property *cardinality-Temperature* specifies that, the number of elements that can be bound is minimum zero (0) and maximum three (3); and, bindings have to define the minimum temperature ($minTemp{>}0°C$) and maximum temperature ($minTemp{<}30°C$), to close and open respectively the automatic window.

3.3 Fine-Feature Configurations

Fine-feature configurations allow product designers to establish bindings among elements and features. Thus, while a constraint is expressed between metaconcepts and features, a binding is expressed between model elements and features. A binding $i = (m, F, R)$ is a tuple composed by a model element m, a feature F, and a logical expression R used to add semantic to the binding.

Figure 3 presents a fine-feature configuration with two bindings, `Binding1`= (`firstFloor`, `airConditioning`, $\{\}$), and `Binding2`= (`mainRoomWindow`, `automaticWindow`, R), where $R = \{maxTemp = 25$ *and* $minTemp = 15\}$.

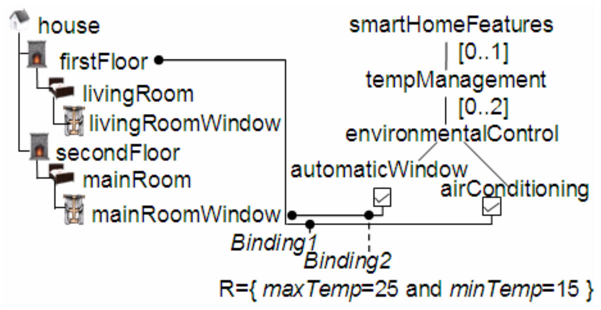

Fig. 3. Fine-Feature Configuration.

The `Binding2` expresses that the `mainRoomWindow` has an `automaticWindow` that has to be opened when the temperature of the room (`mainRoom`) reaches the maximum of $25°C$, and closed when the temperature reaches the minimum of $15°C$. To create constraint models and fine-feature configurations we have extended the Czarnecki et al. feature metamodel.

3.4 Bindings Vs. Constraints

We say that a binding $i = [m, F, R]$ *fits* a constraint $o = [M, F, P]$ when m conforms to M. We note this relationship $i \xrightarrow{f} o$. Therefore, since `mainRoomWindow` conforms to `Window`, then $\texttt{Binding2}\xrightarrow{f}\texttt{ConstraintAutoWindow}$. We automatically validate fine-feature configurations against constraint models. This ensures that each binding i *fits* a constraint o, and satisfy the respective property P. Thus, fine-feature configurations are valid configurations inside the scope of the product line, representing products that include fine-grained variations.

4 Deriving Products

We configure products using feature models and fine-feature configurations. This allows us to express coarse- and fine-grained variations respectively. During the activity of product derivation, we execute successive *model transformations*, starting from domain models, until we derive particular products. We call a *workflow* the successive model transformations we execute. We call *model transformations*, programs that transform source models into target models by using sequences of *transformation rules*.

In our approach, we identify two types of transformation rules. We call *common-transformation rules*, the rules we use to generate the common characteristics of products; and, we call *fine-transformation rules*, the rules we use to generate the fine-grained variations of products. Thus, to generate a base product without including fine-grained variations, we execute a workflow including model transformations that use sequences of common-transformation rules. To generate a product including fine-grained variations, we adapt common-transformation rules in model transformations. Adaptations allow us to derive products including fine-grained variations. For instance, we can conditionally avoid the execution of a common-transformation rule, or to execute a fine-transformation rule *after* or *before* a common-transformation rule is executed.

A recent work [5, 7] has shown that model transformation programs can be modularly adapted based on AOP principles as introduced by AspectJ [1]. Using the same principles, we adapt model transformations by applying aspects to model transformations. An aspect specifies the common-transformation rules to intercept (pointcuts), and the fine-transformation rules to execute (advices).

We use *transformation interceptors* to intercept model transformations and apply aspects to common-transformation rules. We make conditional the execution of transformation interceptors by using *flow conditioners*. A flow conditioner determines when a transformation interceptor has to be executed, according to the current configuration (fine-feature configuration) of a product.

Figure 4 presents a workflow example (`SmartHomeWF`) that makes possible the derivation of Smart Home Systems based on a fine-feature configuration (`Fine-FeatureModel`).

The flow conditioner (`Conditioner`) checks if exist bindings in the `Fine-FeatureModel` that fit a specific constraint. If any, the transformation interceptor (`Interceptor`) is executed. `Interceptor` intercepts the model transformation `WindowsModelTransf`, which transforms `Window` elements, and execute the aspect `AutoWindowAspect`. The aspect specifies the pointcut, in this case the common-transformation rule `Gen-TransRule-2`, and the advice, in this case the fine-transformation rule `F-AutoWindow`.

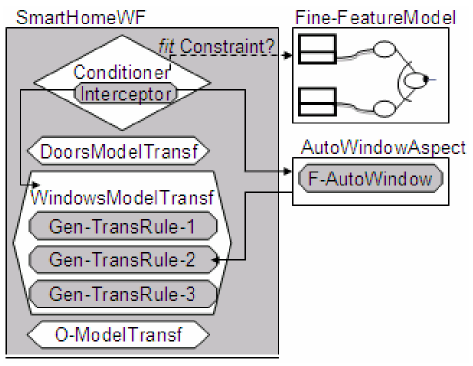

Fig. 4. Workflow Example.

4.1 openArchitectureWare

To implement our approach, we have used and extended the openArchitecture-Ware toolkit (oAW). oAW is a MDD framework integrated into Eclipse. oAW integrates facilities not only to transform models-to-models, but also to transform models-to-text (source code). At the core of oAW, there is a workflow engine allowing the definition of transformation *workflows* by sequencing diverse *workflow components*. oAW has some pre-built workflow components that facilitate the reading and instantiation of models, checking them for constraint violations, and transforming them into other models or source code. Transformation and generation workflows are built by using XML files that describe the steps needed to be executed in a generator run. Each of these steps is specified as a workflow component.

In oAW, model-to-model transformations are implemented using a language called Xtend. It is a textual functional language for querying and navigating existing models as well as building new models.

oAW provides support for AOSD. There is a workflow component that facilitates the definition of transformation interceptors. This is called the *transformationAspect* component. Furthermore, Xtend includes support for describing aspects that use *around* advices on transformation rules.

Listing 1 shows a workflow example. First, an EMF model (`domainModel.xmi`) is read. A workflow slot `source` is used to send the `domainModel` to the transformation component. Next, the model is transformed by invoking a model transformation (line 9) that starts with the execution of the transformation rule `createSimpleWindows`. The transformed model is available for further processing in the `target` slot.

```
<workflow>
 <component id='xmiParser'
  class='org.openarchitectureware.emf.XmiReader'>
  <modelFile value='domainModel.xmi'/>
  <outputSlot value='source'/>
 </component>

 <transform id='XtendComponent.model2model'>
  <invoke value='createSimpleWindows(\${source})'/>
  <outputSlot value='target'/>
 </transform>
</workflow>
```

Listing 1. oAW Workflow Example

4.2 Loading Fine-Feature Configurations

To create a SPL including fine-grained variations, we create a general constraint model. We configure particular products including fine-grained variations creating fine-feature configurations. Using oAW, we create a workflow with scheduled workflow components that execute model transformations; and, we create common-transformation rules using Xtend.

Since we apply advices to common-transformation rules only in the case we find bindings that fit specific constraints, we need to query the fine-feature configuration at any moment of the oAW workflow execution. To perform this, we have created an oAW component that makes possible to load a constraint model and a fine-feature configuration into the execution context of an oAW workflow. Thus, a constraint model and a fine-feature model have to be available to be queried during the execution of the workflow. We call this component, the *fine-configuration loader* component.

4.3 Intercepting Workflows Execution

During the definition of oAW workflows, we use the oAW $transformationAspect$ component to define transformation interceptors. As we introduced above, we make conditional the execution of transformation interceptors using flow conditioners. A flow conditioner indicates when a transformation interceptor has to be executed, according to the current fine-feature configuration loaded in the context of the oAW workflow execution.

We have created an oAW component to define flow conditioners. We call this, the *fine-feature* component. This component allows us to query a loaded fine-feature configuration to know if there are bindings that imply the adaptation of particular model transformations. For example, assume that we have a binding i such as, $i \xrightarrow{f}$`ConstraintAutoWindow`. In this case, a model transformation transforming elements that conform to `Window` must be intercepted. After the interception is done, the model transformation can be adapted to transform the elements involved in such bindings (elements that conform to `Window`) into `automaticWindows`.

Listing 2 shows a workflow example using the *fine-feature* component. The component queries a loaded fine-feature configuration searching bindings $i = (m, F, A)$ such that, m conforms to the `Window` metaconcept (line 4) and F is the feature `automaticWindow` (line 5). If any, the (oAW component) transformation interceptor `transformationAspect` (line 6) is executed. This indicates that the model transformation `XtendComponent.model2model` has to be intercepted (line 7), and the aspect `extensionAdvice` (line 8) has to be executed.

```
   <workflow>
 2  <component id='fineFeature'
      class='org.oAW.fineVariation.FineFeature'>
 4    <isBoundAny value='Window'/>
      <toFeature value='automaticWindow'/>
 6    <transformationAspect adviceTarget=
        'XtendComponent.model2model'>
 8      <extensionAdvice value='extensionAdvice'/>
      </transformationAspect>
10  </component>

12  <transform id='XtendComponent.model2model'>
      ...
14  </transform>
   </workflow>
```

Listing 2. Fine-Feature Component Example.

4.4 Using Fine-Transformation Rules

To create products including fine-variations, we adapt the execution of common-transformation rules by using aspects. We create aspects taking advantage of the facilities included in the (oAW) Xtend language. In particular, the pointcuts we define match execution points of common-transformation rules, and the advices we apply are fine-transformation rules.

Since Xtend facilitates the definition of *around* advices, once a common-transformation rule is intercepted, we can decide either to avoid its execution, or to execute it before or after our fine-transformation rule.

A fine-transformation rule transforms models to include fine-grained variations. This rules query fine-feature configurations to know the specific elements of a model, involved in bindings, that have to be transformed to include fine-grained variations. For example, assume the case where a $Binding2 =$ (`mainWindow`, `automaticWindow`,$\{\}$) $fits$ the `ConstraintAutoWindow` (see Figure 3). A fine-transformation rule queries the fine-feature configuration and identifies the `mainWindow` element. Then, the rule transforms the `mainWindow` into an `automaticWindow`.

Listing 3 presents an example of an Xtend aspect and a fine-transformation rule. In the aspect, the common-transformation rule `createSimpleWindows()` is intercepted (line 1) and the fine-transformation rule `fineRule()` is executed (line 2). Note that in this example, the common-transformation is avoided. Then, `fineRule()` executes a function `fit(String)`, which returns the model elements involved in bindings that fit the `ConstraintAutoWindow` (line 5). Finally, each of such elements is transformed into elements of the target model representing `automaticWindows` (line 6).

```
around createSimpleWindows():
  fineRule(f, res);

private fineRule():
 let windows = fit('ConstraintAutoWindow')):
 windows.createAutomaticWindow;
```

Listing 3. Example of Xtend Aspect and Fine-Transformation Rule.

5 Conclusions

In this paper, we introduced an approach to define fine-grained variations of SPLs while ensuring consistency between variation points using an aspect- model-driven approach. Our proposal is based on feature modeling, meta-modeling and AOP principles. We have used Domain Specific Modeling to address the problem of expressing SPL variability. Thus, we allow the easy configuration of products done by product designers who are experts in specific domains.

We have expressed fine-variations by using fine-feature configurations, and we have constrained such variations creating constraint models. Thus, we achieved to express fine-grained details of product configurations, included in a well defined scope of a SPL.

In SPL engineering, problem space focuses on defining what problem the family of applications, or an individual application in the family, will address. The solution space focuses on producing the software components to solve that problem. Our approach narrows the gap between the problem and the solution space deriving products by means of automatic model transformations. We derived products that include fine-grained variation crosscutting common-transformation rules with fine-transformation rules. For this, we have used AOP applied in the context of model transformations to derive products where variability has been bound to specific model elements.

References

1. AspectJ website: AspectJ website http://www.eclipse.org/aspectj.
2. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. Volume 1241 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (jun 1997) 220–242
4. Third Workshop on Models and Aspects Handling Crosscutting Concerns in MDSD, ECOOP. (2007)
5. Voelter, M.: Patterns for Handling Cross-cutting Concerns in Model-Driven Software Development. In: 10th European Conference on Pattern Languages of Programs (EuroPLoP). (2005)
6. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)

7. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proceedings of the 11th International Software Product Line Conference. (2007) 233–242
8. C-SAW website: C-SAW website http://www.cis.uab.edu/gray/ Research/C-SAW/.
9. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley, Amsterdam (2005)
10. Groher, I., Voelter, M.: Expressing Feature-Based Variability in Structural Models. In: Workshop on Managing Variability for Software Product Lines. (2007)
11. Groher, I., Voelter, M.: Xweave: models and aspects in concert. In: AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling, New York, NY, USA, ACM (2007) 35–40
12. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Glück, R., Lowry, M.R., eds.: GPCE. Volume 3676 of Lecture Notes in Computer Science., Springer (2005) 422–437
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Proceedings of the Third Software Product Line Conference 2004, Springer, LNCS 3154 (2004) 266–282
14. Garces, K., Parra, C., Arboleda, H., Yie, A., Casallas, R.: Variability Management in a Model-Driven Software Product Line. Avances en Sistemas e Informática **4**(2) (September 2007) 3–12
15. openArchitectureWare (oAW) website: http://www.openarchitectureware.org.
16. European Commission STREP Project AMPLE IST-033710: http://ample.holos.pt/.
17. Eclipse Modeling Framework (EMF) website: http://eclipse.org/emf.
18. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 (1990)