Aspect-Oriented Model-Driven Software Product Line Engineering

Iris Groher and Markus Voelter

SEA Institute, Johannes Kepler University Linz, Austria,
Independent Consultant, Goeppingen, Germany
{iris.groher@students.jku.at,voelter@acm.org}

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. This article presents an approach that facilitates variability implementation, management, and tracing by integrating model-driven and aspect-oriented software development. Features are separated in models and composed by aspect-oriented composition techniques on model level. Model transformations support the transition from problem to solution space models. Aspect-oriented techniques enable the explicit expression and modularization of variability on model, template, and code level. The presented concepts are illustrated with a case study of a home automation system.

Key words: Software product line development, aspect-oriented software development, model-driven software development

1 Introduction

Most high-tech companies provide products for a specific market. Those products usually tend to have many things in common. An increasing number of these companies realize that product line development [41] [10] fosters planned reuse at all stages of the lifecycle, shortens development time, and helps staying competitive.

Commonalities between products in the portfolio as well as the flexibility to adapt to different product requirements are captured in so called *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 defined as an increment in functionality provided by one or more members of a product line [6]. 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. Variability of features often has widespread impact on multiple artifacts in multiple lifecycle stages, making it a pre-dominant engineering challenge in software product line engineering.

Despite their crucial importance, features are rarely modularized and there is only little support for incremental variation of feature functionality. The reason is that feature-specific parts are often of crosscutting nature. On implementation level often pre-processors are used to wrap feature-specific code fragments in #if-#endif statements. Listing 1 shows an implementation example of the eCos operating system [16]. The `Cyg_Mutex` constructor includes 29 lines of code. Four lines of code implement the actual business logic (lines 1, 3, 4, and 29), two lines set the tracing policy (lines 2 and 28) and 23 (almost unreadable) lines implement optional features.

```
1  Cyg_Mutex::Cyg_Mutex() {
       CYG_REPORT_FUNCTION();  //tracing policy
3      locked = false;
       owner = NULL;
5  #if defined(CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT) &&
         defined(CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
7  #ifdef CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
       protocol = INHERIT;
9  #endif
   #ifdef CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
11     protocol = CEILING;
       ceiling = CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRI;
13 #endif
   #ifdef CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
15     protocol = NONE;
   #endif
17 #else // not (DYNAMIC and DEFAULT defined)
   #ifdef CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_CEILING
19 #ifdef CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
       // if there is a default priority ceiling defined, use that
21     // to initialize the ceiling.
       ceiling = CYGSEM_KERNEL_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
23 #else
       ceiling = 0; // Otherwise set it to zero.
25 #endif
   #endif
27 #endif // DYNAMIC and DEFAULT defined
       CYG_REPORT_RETURN();   //tracing policy
29 }
```

Listing 1. eCos implementation example [31]

Another pitfall is that features and architecture are often derived from requirements in a non systematic, ad hoc way. For software product lines it is essential to know the relationship between requirements, the derived architecture, the design and the implementation artifacts. Consequently, a systematic way to group requirements into features that are then related to architectural entities and a seamless tracing of requirements throughout the whole lifecycle is necessary.

As demonstrated, variability tends to crosscut multiple points in code as well as different other artifacts in the software development lifecycle. Moreover, the effects of variability and, in particular, new variations brought in by evolution,

tend to propagate in ways that cannot be easily modeled or managed. New requirements may necessitate changes to code, design, documentation, and user manuals amongst many other artifacts and assets that go into making a product line. Ensuring the traceability of requirements and their variations throughout the software life cycle is key for successful software development general and a successful product line in particular.

Also the mapping between problem space and solution on space is not trivial. The problem space is concerned with end-user understandable concepts representing the business domain of the product line. The solution space deals with the elements necessary for implementing the solution, typically IT relevant artifacts. There is a many to many relationship between entities in the problem space (requirements and features) to entities in the solution space (software components).

Aspect-oriented software development (AOSD) [18] [29] improves the way software is developed by providing means for modularizing crosscutting concerns. They are encapsulated as aspects and powerful mechanisms support their subsequent composition with other software artifacts. Aspects interact with other artifacts at so called *join points*, well defined points in the structure or execution flow of an artifact or a program. Pointcut expressions quantify over the join points to select the set of actual composition points for a specific aspect. An aspect weaver automatically composes aspects with the rest of the system, either statically during compilation, dynamically at runtime, or at load-time.

Model-driven software development (MDSD) [44] [21] improves the way software is developed by capturing key features of a system in models which are developed and refined as the system is created. During the system's lifecycle, models are synchronized, combined, and transformed between different levels of abstraction and different viewpoints. In contrast to traditional modeling, models do not only constitute documentation but are processed by automated tools. Thus models have to be formal. Every model is an instance of a meta model. The meta model defines the vocabulary and grammar, i.e. the abstract syntax used to build models. In order to be useful for MDSD, models have to be complete regarding the abstraction level or viewpoint they describe.

While AOSD and MDSD are different in many ways – MDSD adds domain-specific abstractions and AOSD offers improved support for concern modularization across the life cycle as well as powerful composition mechanisms – they also have many things in common, e.g., they help the developer to reason about one concern at a time. Essentially, AOSD and MDSD complement each other.

We propose an approach that facilitates variability implementation, management, and tracing from architectural modeling to implementation of product lines by integrating both AOSD and MDSD. When building product lines, our integrated approach increases 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 space 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.

The presented concepts are illustrated with a case study of a home automation system. The case study is based on real-world system requirements from Siemens AG and demonstrates the benefits of the presented approach. We used data from the case study to answer our research question:

> *Where and how can software product line development benefit from AOSD, MDSD, and their combination?*

The remainder of this article is organized as follows: Section 2 provides an overview of our integrated aspect-oriented model-driven software product line development approach. Additionally, it gives an overview of openArchitectureWare, the MDSD framework our tools are based on. Section 3 illustrates the selective adaptation of models and provides an overview of XWeave and XVar. AO on model transformation level is discussed in Section 4. Section 5 discusses AO on code generation level. Variability on code level is addressed in Section 6. We report on the case study we conducted at Siemens in Section 7. We conclude with a summary and a note on future work in Section 8.

2 Aspect-Oriented Model-Driven Software Product Line Engineering

This section gives an overview of what we call *Aspect-Oriented Model-Driven Software Product Line Engineering (AO-MD-PLE)*. The key parts are presented in this section, while the separate building blocks and tools are demonstrated in detail in the subsequent sections.

Our approach [48] integrates AO and MDSD into product line development to facilitate variability implementation, management and tracing from architectural modeling to implementation. We argue that due to the fact that models are more abstract and hence less detailed than code, variability on model level is inherently less scattered and therefore simpler to manage (cf. Figure 1). Variability can be described more concisely since in addition to the traditional mechanisms (e.g. patterns, frameworks, polymorphism), variability can be described on the more abstract level of models.

AO-MD-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.

The approach we propose is as follows:

- We express as many artifacts as possible using models, i.e. we give them representations on model level. This allows for processing these artifacts using model transformations.

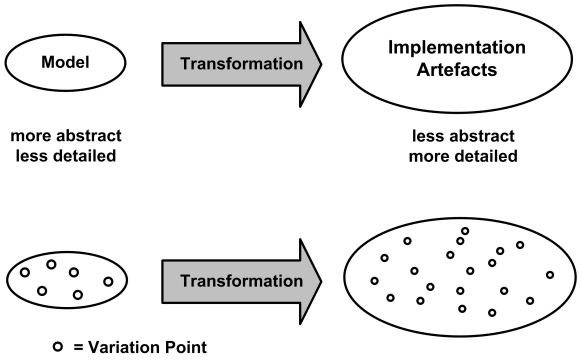

Fig. 1.
Mapping abstract models to detailed implementations

- Mappings from problem to solution space are implemented as model-to-model transformations. This enables to formally describe the mappings and automate their execution.
- Variable parts of the resulting system are assembled from pre-built assets generated from models. This is more efficient and less error prone than manual coding.
- Aspect-oriented modeling (AOM) [9] [5] is used to implement variability in models. This supports the selective adaptation of models.
- AO on model transformation and code generation level [47] is used to implement variability in transformers and generators.
- Aspect-oriented programming (AOP) [28] [4] is used to implement crosscutting features on code level that cannot be easily modularized in the generator.
- Certain parts of the product will still be implemented manually because, for economic reasons, developing a custom generator is too costly. The manually written code is integrated with the generated code in well-defined ways.

An overview of AO-MD-PLE is given in Figure 2. Domain requirements are captured in a problem space meta model. Based on product requirements, a problem space model is created that is an instance of the problem space meta model. The problem space meta model defines the vocabulary and grammar, i.e. the abstract syntax, used to build the problem space model. The model itself is built using a Domain Specific Language (DSL) [44]. A DSL is a formalism for building models. It encompasses a meta model (in this case the problem space meta model) as well as a definition of a concrete syntax that is used to represent the model. The concrete syntax can be textual, graphical, or using other means such as tables, trees, or dialogs. It is essential that the concrete syntax can sensibly represent the concepts the DSL is intended to describe. A suitable editor has to be provided that supports the creation of models using the DSL.

Both, problem space meta models and problem space models can be configured using either model weaving or model tailoring. In model weaving, optional

parts are added to a minimal core whereas in model tailoring optional parts are removed from an overall model.

The tool developed to support model weaving is called XWeave and the tool to support model tailoring is called XVar. Both allow the selective adaptation of models and meta models based on feature selections in configuration models. The model configuration approach and the respective tools will be introduced in great detail in Section 3.

A formal mapping is defined between the problem space meta model and the solution space meta model. The defined mapping allows for an automatic transformation of the problem space model into the solution space model. This step can be repeated as desired. For example, one could first map the problem space meta model to a platform independent model and map this meta model to a platform specific model. This separation makes it possible to easily support different platform technologies. Model transformations can be configured using AO on transformation level.

We used an AO model transformation language called Xtend that supports the selective adaptation of transformation functions. Again, model transformation aspects can be linked to features defined in a feature model. AO on transformation level and the respective language will be illustrated in Section 4.

In order to create a running system, code is generated from the solution space model. This step can also be configured using AO on template level and is supported by a language called Xpand. Code generation template weaving will be described in Section 5. As a 100% generation is not realistic, manually written code has to be integrated with the generated code in well defined ways. Also, it might be necessary to integrate pre-built reusable components in the form of libraries into the generated system. Due to the fact that in our approach all artifacts have representations on model level, we can process them using model transformations. Based on the information in the models, we can determine whether a given component (either manually written or pre-built) is part of a product and thus has to be integrated.

Due to the fact that it is in most cases necessary to include manually written code, variability on source code level is an issue. We use aspect-oriented programming (AOP) to implement positive variability on code level. Additionally, we developed a tool called XVar that supports negative variability on code level. We will elaborate on this in Section 6.

An important concern in software product lines is tracing, as stakeholders want to be able to trace how a given requirement results in a certain software configuration. Using our AO-MD-PLE approach this is relatively easy to do. Since mappings between abstraction levels are based on formal model transformations, we can make sure the mappings are made persistent in a trace model. It can either be built automatically by the transformation engine, or it can be built manually by the transformation developer.

Since we trace between model elements, the trace is finer grained than in current approaches where tracing happens between artifacts [33] [27]. Also, since problem space concepts are also represented as models and these models have a

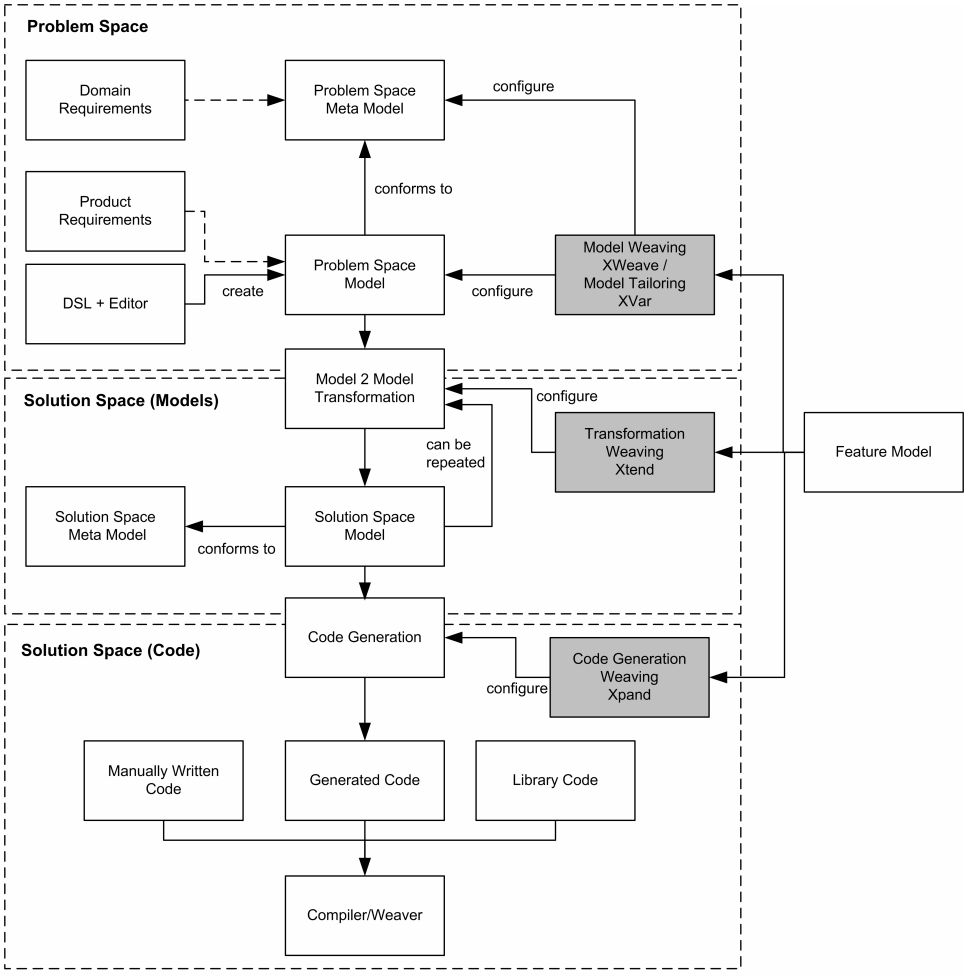

Fig. 2. Aspect-oriented model-driven product line engineering

defined mapping to solution space models, we gain traceability from the problem space over to the solution space.

Additionally, tracing down to code level is an important issue. Specific regions of code need to be associated with model elements. For generated code this is straight forward since the generator knows which model elements are "in scope" when a given region of code is generated. For manually written code it is more challenging, since a piece of hand-written code may implement any number of requirements. This problem can be mitigated to some extend by clearly defining the locations where manually written code can be integrated. An alternative approach is to specify the trace from models to code manually. Tracing to library code is again relatively easy to do in our approach as every library component has some kind of representation on model level. We can then trace via model element relationships.

On the other end of the spectrum we also need to trace requirements. These are different from problem space models as requirements are typically plain English text. To make them traceable, we need to somehow integrate them into the "modeling world". This can be done in various ways, depending on the tool that is used to capture the requirements. For example, it would be possible to create an EMF model [14] based on requirements managed with the DOORS [46] tool and use the EMF model for tracing purposes.

2.1 openArchitectureWare

This section introduces openArchitectureWare (oAW) [38] [39], a toolkit for all aspects of MDSD. The tools developed in the course of our work are all part of the oAW framework.

oAW is an open source MDSD framework implemented in Java and integrates a number of tool components. oAW supports arbitrary import model formats, meta models, and output code formats. oAW is integrated into Eclipse and provides various plugins that support model-driven development. It contributes to and reuses components from the Eclipse Modeling Project [14].

At the core, there is a workflow engine allowing the definition of transformation workflows by sequencing various kinds of workflow components. oAW comes with pre-built workflow components for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code. oAW provides a family of specialized languages for specifying constraint checks, transformations and code generators. All of those languages are built on a common OCL-like expression language. Editors and debuggers integrated into Eclipse are provided for all those languages.

Figure 3 gives an overview of oAW and its main building blocks. The following list explains the different components (marked with numbers):

1. Model verification using constraints: Both models and meta models can be checked for validity. In oAW constraints are defined using the Checks language.

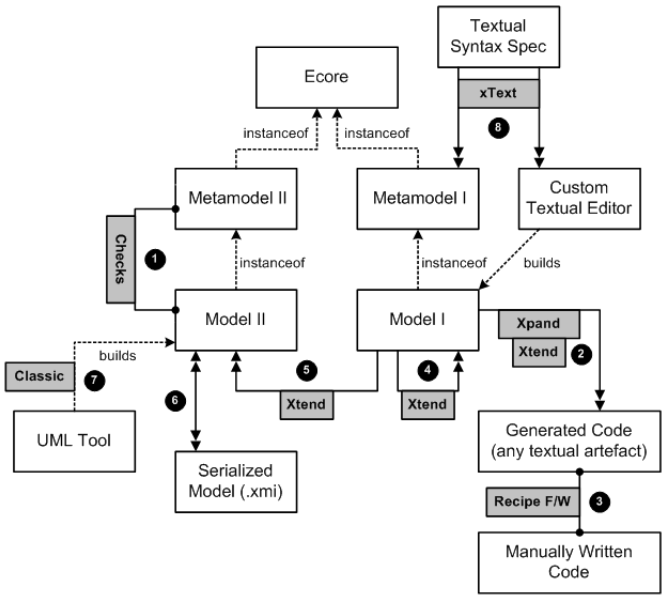

Fig. 3. Overview of openArchitectureWare [38]

2. Artifact generation: Any textual artifact can be generated from models using the Xpand template language.
3. Integrating generated code with manually written code: The Recipe Framework can be used to enforce rules on the code base. For example, a generator can generate an abstract base class from which developers extend their implementation classes containing the business logic. Such rules can be specified using Recipes.
4. Model modification: Models can be modified or completed using the Xtend language.
5. Model transformation: Models can be transformed into other models using the Xtend language. Typically, input and output models are instances of different meta models.
6. Loading and storing models: By default, EMF models are stored in XMI [36] files. oAW provides workflow components for loading and storing models from/to XMI files.
7. Model creation using UML tools: Models can be created and edited using familiar UML tools such as Rational Rose [25] or MagicDraw [35]. oAW provides adapters for various commonly used UML tools.
8. Textual model creation: oAW can also read textual models created with custom-built textual editors based on its Xtext framework. The DSL is described in EBNF and the parser, meta model, and customized editor is created automatically.

9. oAW also integrates with custom graphical editors built using Eclipse GMF [15]. oAW constraints can be evaluated inside GMF editors in real time, and oAW can process models created with GMF.

The transformation and generation workflow is described using the oAW workflow language. As of version 4.2, 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 is called 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.

Listing 2 shows an example workflow. First, an EMF model (`inputModel.xmi`) is read from its XMI-based serialization format. The top-level meta model package is `data.DataPackage`. A workflow slot `model` is used to pass the instantiated model to the transformation component. Slots are named variables global to the workflow. Next, the model is transformed by invoking a transformation function `transform` on the model. The transformed model is available for further processing in the `transformedModel` slot.

```
<workflow>
   <component id="xmiParser"
      class="org.openarchitectureware.emf.XmiReader">
      <modelFile value="inputModel.xmi" />
    <metaModelPackage value="data.DataPackage" />
    <outputSlot value="model"/>
  </component>

   <transform id="XtendComponent.model2model" >
      <invoke value="model2model::transform(\${model})"/>
      <outputSlot value="transformedModel"/>
   </transform>
</workflow>
```

Listing 2. oAW example workflow

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. Listing 3 shows an example transformation in which an `Interface` is transformed into a `Service`.

```
create osgimm::Service Interface2Service( cbdmm::Interface intf ):
   setName( intf.name ) ->
   setOperations( (List)intf.operations.clone() );
```

Listing 3. oAW example Xtend transformation

Code generation is done using a language called Xpand. It is an object-oriented template language that supports polymorphism. Listing 4 shows an example Xpand template file. First, the meta model is imported. Second, a new file with the name of the entity `Class` including the class skeleton is generated. The template «`name`» is substituted with a concrete value (the name of the class) at generation time.

```
«IMPORT metamodel»

«DEFINE javaClass FOR Class»
    «FILE name + ".java"»
```

```
        public class <<name>> {
            // add implementation of class here
        }

    <<ENDFILE>>
<<ENDDEFINE>>
```

Listing 4. oAW example Xpand template

Each oAW language (Check, Xtend, Xpand) is based on a common expression language and type system. This has the advantage that all the languages can operate on the same models, meta models, and meta meta models, have a common look and feel to them and are therefore easy to learn. The expressions framework provides a uniform abstraction layer over different meta meta models [39].

Listing 5 gives an overview of the oAW expression language. It shows how properties and operations can be accessed and how types and static properties are evaluated. The expression language offers the built-in types `Object`, `Void`, `String`, `Boolean`, `Integer`, and `Real`. The current object can be accessed using the `this` variable. There are special operations for collections that evaluate expressions on elements within the collection. Chain expressions support the sequence evaluation of several expressions. Local variables can be defined using the let expression. The oAW expression language is statically typed.

```
/* accessing a property */
modelElement.someProperty
/* accessing an operation */
modelElement.someOperation()

/* evaluates to the type 'a::specific::Type' */
a::specific::Type
/* evaluates the static property 'a::specific::PROPERTY' */
a::specific::PROPERTY

/* returns elements of collection for which boolean-expression
   evaluates to true */
collection.select ( v | boolean-expression )

/* returns elements of collection for which expression evaluates
   to false */
collection.reject ( v | boolean-expression )

/* returns elements of collection that are instances of a
   specific class */
collection.typeSelect ( classname )

/* returns elements of collection where expression is
   evaluated for each element */
collection.collect ( v | expression)

/* boolean-expression has to be true for all elements of
   collection */
collection.forAll ( v | boolean-expression )

/* boolean-expression has to be true for at least one element
   of collection */
collection.exists ( v | boolean-expression )

/* conditional expressions */
condition ? thenExpression : elseExpression
```

```
/* chain expression */
firstExpression -> secondExpression -> thirdExpression

/* definition of local variables */
let v = expression in expression
```

Listing 5. Overview of the oAW expression language

Integrating oAW and pure::variants This section illustrates how the variant management tool pure::variants [42] has been integrated into oAW to enable seamless and efficient aspect-oriented model-driven product line development.

pure::variants is a variant management tool that manages product line variability and assists in managing and assembling individual product variants. The basic idea of pure::variants is to realize product lines by feature models and family models. The problem space is captured with feature models and the solution space is captured with family models, separately and independently. A family model consists of so-called components. Components represent at least one attribute of the software solution and contain the logical parts like classes, objects, functions, variables, and documentation. A feature model specifies the interdependencies of the features of a product line. They represent all variability of the products in the product line. pure::variants also supports the use of multiple feature models that are hierarchically linked.

Users can select features required for the desired product from the feature models. A configuration model represents such a selection of features. pure::variants checks the validity of this selection and if necessary automatically resolves dependency conflicts. A valid selection triggers an evaluation of the family models that contain component definitions consisting of logical and physical parts. The evaluation process results in an abstract description of the selected solution in terms of components.

Within the oAW tooling, global configuration models can be queried and hence, MDSD activities can be performed based on the selection of features. Workflow components, model-transformations, and code generation templates can be linked to features. Their execution then depends on the presence or absence of features in the configuration model.

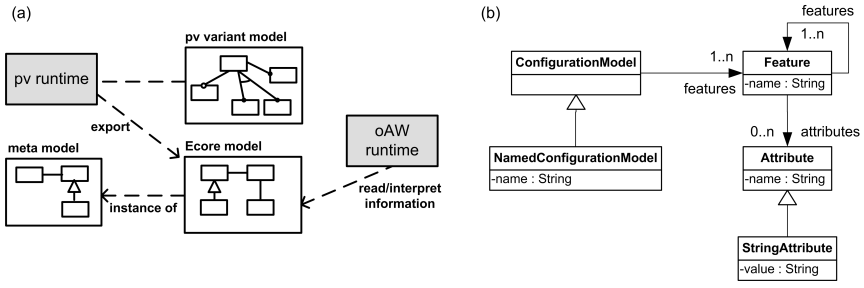

Fig. 4. Integration of pure::variants and oAW

As illustrated in Figure 4 (a), the data transfer between pure::variants and oAW is done using EMF Ecore. An automatic variant model export has been integrated into pure::variants. The oAW runtime reads this model and makes the variant information available for querying by oAW workflows, transformations and templates. In order for this integration to work, a meta model for configuration models (Figure 4 (b)) has been defined. A `ConfigurationModel` contains a list of `Feature`s that again contain features (subfeatures). Features can contain attributes that have a name and a value.

The dependency between workflow steps and features is expressed by the surrounding < feature ... > tag. Listing 6 shows an example. If and only if the feature `debug` is selected in the global configuration model, code that implements the `debug` feature is generated.

```
<feature exists="debug">
   <component class='oaw.xpand2.Generator>
      ... invoke generator that generates debugging code
   </component>
</feature>
```

Listing 6. Dependency between workflow and features

It is also possible to access the configuration model directly from within transformations or code generation templates. In Listing 7 the transformation function `handleDebugFeature` is only called if the `debug` feature is selected in the configuration model.

```
create Service transformInterface2Service (Interface f) :
   ...
   hasFeature("debug") ? handleDebugFeature() -> this ) : this;

handleDebugFeature (System sys) :
  setValue( (String) getFeatureAttributeValue ("debug", "level") ) ->
  ...
```

Listing 7. Dependency between transformation and features

The integration of pure::variants and oAW also supports addressing properties or attributes of features. The values of properties and attributes can be read and used in the transformation or code generation templates. In Listing 7 the debug level is read and handled within the transformation.

The integration of variant management tools other than pure::variants is easily possible. As long as the tool provides an export of variant information to the defined meta model in Figure 4, oAW can read and interpret this information. The variant information is then automatically available to the workflow, transformation, and generation engine.

3 Expressing Variability in Structural Models

This section describes concepts and tools that support the definition of feature-based variability in structural models and hence the selective adaptation of models. Structural models are models built with a creative construction DSL. Features expressed in structural models can be linked to configuration models. This

enables the adaptation of those structural models based on the feature selection in configuration models.

3.1 Positive Variability

The terms positive and negative variability have been initially introduced in [11]. As illustrated in Figure 5, positive variability starts with a minimal core and selectively adds additional parts. The core represents the model parts that are common to all products within the product line. Varying model parts are attached to the core based on the presence or absence of features in the configuration models.

Negative variability selectively takes away parts of a creative construction model. It will be discussed in detail in Section 3.2

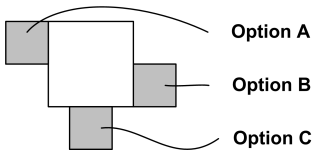

Fig. 5. Positive variability

When expressing variability in models, optional model elements have to be connected to the core at specific points. This is analogous to the concept of join points in aspect-orientation. In AOP, join points are those elements of the programming language semantics which the aspects coordinate with [2]. In modeling, join points are elements of the modeling language which aspects coordinate with. In product lines, variation points represent variability subjects within domain artifacts [41]. Domain artifacts include all kinds of development artifacts such as requirements, architecture, design, code, and tests. This similarity in the concept of a variation point and a join point makes AOM and specifically model weaving well-suited candidates for implementing positive variability in structural models. Similar to weaving of code level aspects in traditional AO languages, aspects are defined on model level and are composed with a base model. Weaving is technically done by an aspect weaver at designated join points.

In the field of AOP, there are various join point models around and many are still under development. Join point models heavily depend on the underlying programming language and AO language [2]. Hence, the join point model of AOM depends on the underlying modeling language and the language used to express aspects on model level. We introduce a model weaving approach including a join point model that supports the expression of aspects on model level and a composition technique for base models and aspect models.

Figure 6 illustrates the concept of model weaving. A given base model (`M A`) and an aspect model (`M Aspect`) are composed. The aspect model consists of pointcut definitions that capture the points where in the base model additional

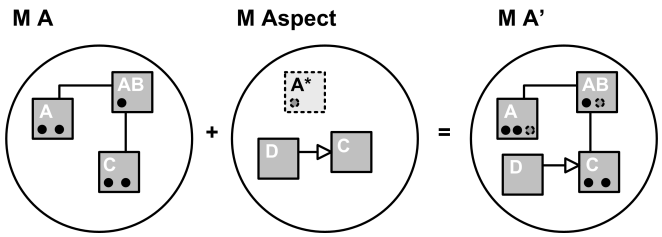

Fig. 6. Model weaving

model elements should be added. In addition to the pointcut definitions, the aspect model contains advices that represent the elements to be added. The aspect model uses both name matching and pointcut expressions to select the desired join points. Both techniques will be explained in great detail later in this section. In this case, the model element `D` of the aspect model `M Aspect` that is derived from element `C` is added to the result model. The weaving rule applied here is simple name matching. The elements named C in both models correspond, so both elements are combined. This results in `C` having a new child `D` in the result model. The aspect element `A*` specifies that the dotted gray element within `A*` should be added to all base elements whose name starts with A, in this case element `A` and element `AB`. This is an application of a pointcut expression. After the weaving process a result model (`M A'`) is created that contains all the original base model elements plus the aspect elements added at the desired join points.

A tool called XWeave[1] [22] [23] has been developed that implements the concepts presented above. It is based on Eclipse as a tool platform, Ecore as the meta meta model. Since it is an oAW workflow component, it easily integrates into oAW based model processing.

XWeave is a model weaver that can weave models that are either instances of Ecore (called meta models) or instances of these meta models (called models). The tool takes a base model as well as one or more aspect models as input and weaves the content of the aspect model into the base model. Weaving an aspect element means that all properties of the element including its child elements are woven into the base model. Both aspect model and base model must conform to the same meta model. The aspect model consists of definitions that capture the points where in the base model the additional model elements should be added (the *pointcut* in AO terms). It also contains these additional model elements (the *advice* in AO terms). This is a form of asymmetric AO. During weaving, aspect elements are physically woven into the base model. The result of this process is an updated model. Subsequent tooling cannot tell the difference between a woven and a non-woven model.

[1] XWeave is part of openArchitectureWare 4.2 and can be downloaded from http://www.openarchitectureware.org/

The join point model of XWeave is based purely on the meta model of the base model and is thus generic. All elements of the base model are potential join points. This means that any model element of an EMF model can serve as a join point. Pointcuts select sets of those elements where additional elements can be attached.

XWeave provides two ways of specifying pointcuts: name matching and explicit pointcut expressions. The next sections will introduce both approaches in detail. For illustration purposes, all examples use the concrete syntax of UML.

Name Matching Name matching means that if a model element in the aspect model has a corresponding element in the base model (both name and type have to be equal), the elements are combined. Hence, matching happens purely based on name and type equivalence. The advice elements defined in the aspect model are added to the base model at the matched points.

Figure 7 shows an example of name matching in XWeave to specify the desired join points. It is an application of meta model weaving. The model in (a) shows the meta model of a simple state machine. The state machine has a list of states linked by transitions. The meta model only supports simple states. A variation of this meta model is to not only support simple states but also dedicated start and stop states. The model in (b) illustrates the aspect model. `StartState` and `StopState` are derived from `State` and should be woven to the base model. The aspect element `State` has a corresponding element in the base model and is therefore the point at which `StartState` and `StopState` are added. The model in (c) shows the result model after weaving.

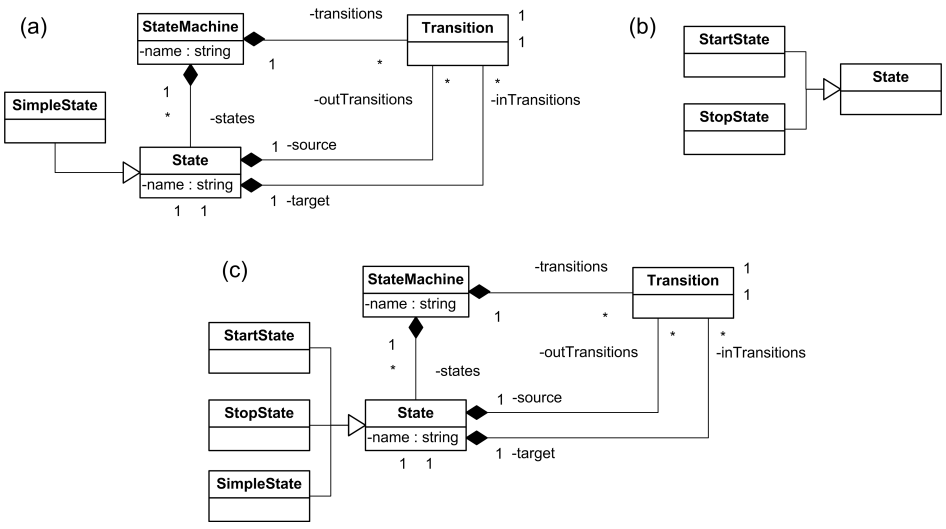

Fig. 7. Meta model weaving using name matching

Pointcut Expressions Another way of specifying pointcuts are explicit pointcut expressions. They can be defined using oAW's expression language. Expressions can select one or more elements of the base model and are defined external to both aspect and base model in a separate expression file. Every expression has a name. The named expressions (pointcuts) can be used in the aspect model. If an aspect element's name starts with %, $, or ? followed by the name of a defined expression, the expression will be evaluated for this element. The % sign introduces an expression that returns a collection of elements. The $ character introduces an expression that returns a single element. The ? character is followed by a String expression, i.e. it has to return a String. It can be used to add any kind of name to the elements. Additionally, it is possible to use * (the asterisk) character as the name of an element. This matches all instances of that particular type.

Pointcut expressions make it possible to select several join points in the model with only one declarative statement. This is referred to as the quantification principle of AO [19]. Expressions must be parameterized with the type that serves as the root for the base model. Whatever the expression returns will be used as the target for the weaving process.

To express pointcuts, XWeave uses the oAW expression language [39] which is a syntactical mixture of OCL and Java. An overview of the most important constructs is given in Section 2.1. A complete reference of the language can be found in [39].

Figure 8 illustrates an example of pointcut expressions. In (a) a state machine model of an oven is presented. It is an instance of the meta model presented in Figure 7. The oven can be either open, closed, or in cooking state. The aspect models in (b) weave an emergency state into the base model including the respective transitions. The aspect model on the left uses a pointcut expression to select all states of type `SimpleState` using the pointcut expression `pc`. The right side of figure (b) is an alternative to using the pointcut expression. The asterisk matches all instances of a particular type, in this case `SimpleState`. The model in (c) shows the result after weaving.

The XWeave approach can be applied to both problem space models and solution space models. This means that both domain models and software models can be configured.

Linking XWeave to Configuration Models Model weaving assists in the composition of different separated models into a consistent whole. It allows to capture feature-dependent parts of models in aspect models. This technique supports a clear separation of variable model parts from the core and supports an automatic composition to create a complete model representing a member of the product line.

During domain engineering the core[2] and the aspects are developed. The core represents model parts common to all products in the product line and the aspects represent features that are increments in functionality provided by one

[2] We will use the terms core and base model as synonyms.

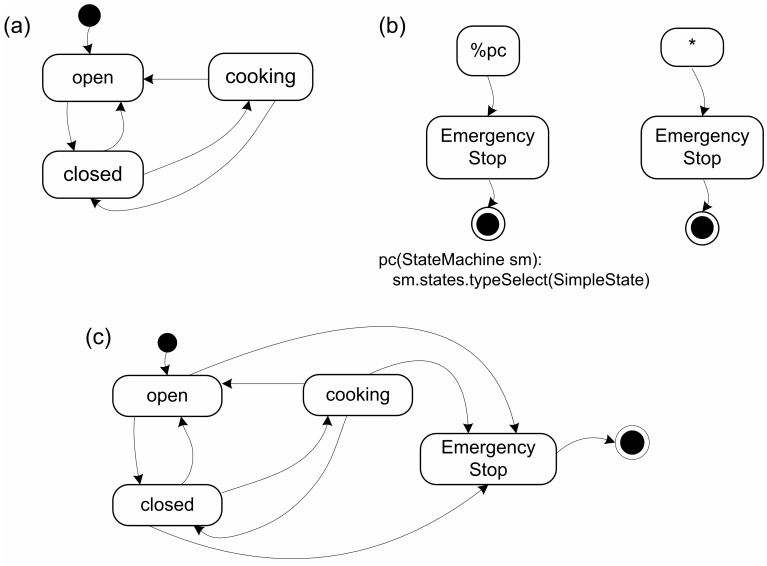

Fig. 8. Model weaving using pointcut expressions

or more members of the product line. During application engineering, the aspect models are composed with the core according to a selection of features in a configuration model. Consequently, the core is minimal in that in only contains elements common to all products in the portfolio. Product-specific parts are added when needed.

The dependency between aspect models and features is specified in the oAW workflow. Aspects that implement optional parts of structural models are linked to features defined in configuration models. Based on a selection of features, the corresponding aspect models are woven to the base model. This dependency between features and aspect models is illustrated in Figure 9.

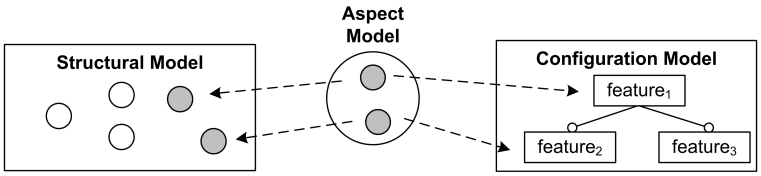

Fig. 9. Linking positive variability to configuration models

Listing 8 shows an example workflow. If the `EmergencyStateFeature` is present in the global configuration model, XWeave weaves the content of the

aspect model into the oven model. Aspect and base model are illustrated in Figure 8. The base model is already in a workflow slot.

```
<feature exists="EmergencyStateFeature">
  <cartridge file="org/openarchitectureware/util/xweave/wf-weave-expr"
    baseModelSlot="ovenModel"
    aspectFile"emergencyStateAspect.xmi"
    expressionFile="expressions"/>
</feature>
```

Listing 8. XWeave workflow

3.2 Negative Variability

As illustrated in Figure 10, negative variability selectively takes away parts of a creative construction model based on the presence or absence of features in the configuration models. This technique is fundamentally different to the technique introduced in the previous section. When using negative variability to implement feature-based variability in structural models, one has to build the "overall" model manually and connect elements to certain features in a configuration model. The model is then tailored based on a certain feature selection, thus model elements are taken away from the full model.

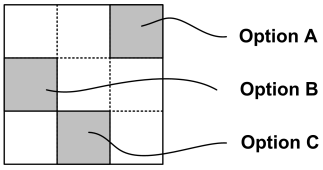

Fig. 10. Negative variability

Figure 11 illustrates how negative variability on model level works. It shows a simple feature model of a person database. A person can either have an international phone number or a local phone number. There are two optional features, namely the possibility of adding state information for US citizens and multiple addresses. Features of the feature model are linked to elements of the structural model below. In this case, attributes and relationships between classes are connected to the features. Those elements are only present in the model if and only if the corresponding features are part of a configuration.

A tool called XVar[3] [22] has been developed that implements the concepts presented above. Like XWeave, it is based on Eclipse as a tool platform, Ecore as the meta meta model, and oAW as the tool for model processing.

Figure 12 illustrates how XVar links structural models to configuration models. A dependency model captures the relationships between model elements and

[3] XVar is part of openArchitectureWare 4.2 and can be downloaded from http://www.openarchitectureware.org/

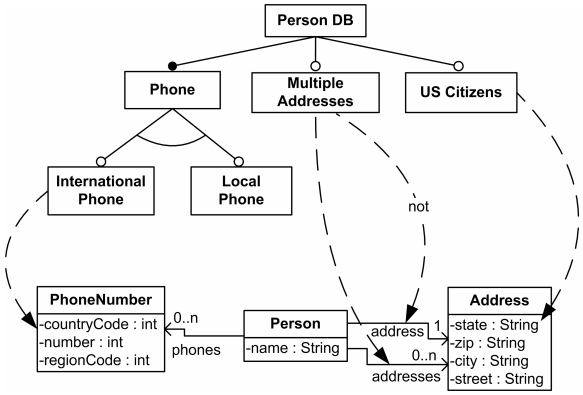

Fig. 11. Negative variability in structural models

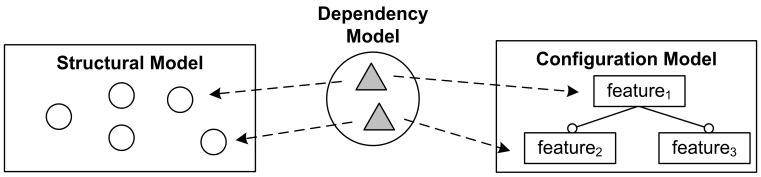

Fig. 12. Linking negative variability to configuration models

features. Depending on the selection of features, the structural model is tailored to only contain the model elements needed for the respective configuration by deleting model elements whose features are not selected. Deleting means that the element itself and all references to this element are removed from the model.

Figure 13 shows the dependency model for the person database example. Every element describes a dependency between a feature and the linked model elements. According to this dependency model, the structural model is tailored. Figure 14 illustrates a model that results if only the optional `US Citizens` and the `Local Phone` features are selected. `International Phone` and `Multiple Addresses` are not part of this configuration. Thus, the attribute `countryCode` of class `PhoneNumber` and the association that supports multiple addresses are deleted from the structural model in Figure 11.

XVar queries the global configuration model and tailors the input model as desired. Listing 9 shows an example workflow. The base model is already contained in a workflow slot.

```
<cartridge file="org/openarchitectureware/util/xvar/emf/wf-xvaremf"
  dependencyFileUri="platform:/resource/personDB/dependencyModel.xmi"
   baseModelSlot="personDBModel" />
```

Listing 9. XVar workflow

XVar can tailor models that are either instances of Ecore (called meta models) or instances of these models (called models). The dependency between fea-

Dependency Model	
Feature Name	**Elements**
International Phone	PhoneNumber : countryCode
US Citizens	Address:state
Multiple Addresses	association addresses
! Multiple Addresses	association address

Fig. 13. Dependency model in XVar

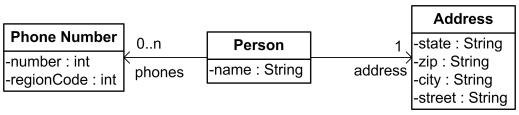

Fig. 14. Resulting model

tures and model elements is specified in a separate dependency model. The advantage of defining feature dependencies in an external model is that no invasive changes to the model are required. All dependencies are explicitly listed in the dependency model. The XVar approach can be applied to both problem space models and solution space models.

3.3 Related Work

Positive Variability in Structural Models The Atlas Model Weaver (AMW) [17] is a tool created as part of the Atlas Model Management Architecture. It's primary goal is to establish links between models. In the first phase of working with AMW, a number of links are established between two or more models. This process can be done manually or semi-automatic. The result is called a weaving model. Based on that model, one can generate model transformations that merge models. AMW is similar to XWeave as both tools can weave or merge models. There is, however, an important difference. AMW contains an interactive tool to build weaving models, whereas XWeave uses name correspondence or pointcut expressions.

C-SAW [20] is a general transformation engine for manipulating models based on aspect specifications using ECL (a variant of OCL). The weaver traverses the model and selects a set of elements to which the aspects should be applied. The advice then modifies the selected element in some way, for example by adding a precondition or changing the element's structure. C-SAW has been developed to tackle the challenge of evolving large models in a consistent way. Instead of applying a set of changes manually, one merely writes an aspect that applies the changes to all selected elements in the model. Comparing it to XWeave reveals that C-CAW does not weave models (in the sense of merging them) as

XWeave does. Rather, it efficiently applies (crosscutting) changes to a collection of elements in a large model.

The Motorola WEAVR [12] is a model weaver developed as a plugin for Telelogic TAU. It supports weaving of UML statecharts that include action semantics and are thus executable. There are two different types of join points, action and transition join points. Advices are encapsulated in a construct called connector. Similar to XWeave, the Motorola WEAVR weaves aspects based on pointcut specifications. The main difference between the two approaches is that XWeave provides a generic EMF based solution that can weave arbitrary models and meta models. The Motorola WEAVR only supports weaving of UML statecharts.

XJoin [39] is a tool for establishing links between meta models. The tool takes two or more existing meta models as input and adds relationships to join them. The partial meta models still keep their own identities and do not need to know about the other ones. Using XJoin different architectural viewpoints [30] can be separately described and later combined. The difference to XWeave is that XJoin does not weave models, it only establishes links between them. Both tools are part of the oAW framework and thus integrate very well.

General AOM approaches [9] are also related to XWeave. Theme/UML [5] for example provides an extension to UML that supports concern modeling and composition. Most AOM techniques are based on UML which is an important difference to XWeave. Also, most approaches lack tool support which is essential for a successful application of the technique in an industrial product line context.

Negative Variability in Structural Models In [13] structural models are connected to feature models to implement negative variability. A feature model is linked to a UML model via stereotypes. Depending on the selected features, the UML model changes. XVar also implements negative variability for structural models but in contrast to [13] it provides a generic EMF based solution. Another important difference is that in XVar the links between model elements and features are managed in a separate dependency model. In [13], the links are managed using stereotypes which requires invasive changes to the model that should be tailored.

In [24] a tool is presented that supports linking features to parts of EMF models. Changes made to structural models are recorded and can be associated to features in the feature model. At product creation time, only the model elements that belong to the currently selected features are present in the model. This approach is similar to XVar in the way how it tailors models according to a specific configuration. An important advantage of XVar is that it models dependencies between features and model elements explicitly in a dependency model. The tool presented in [24] only observes changes to the core but the mapping information is hidden in the tool.

The Gears variant management tool [8] provides a plugin for the Rhapsody modeling tool. Models can become Gears core assets and thus include variation points. Different variants for model elements can be created and are selected

according to a specific configuration. The model does not change, only the generated code varies. XVar actually changes the model which is the main difference to the Gears/Rhapsody bridge. Also, Rhapsody only supports UML models.

4 Expressing Variability in Model Transformations

This section describes concepts and tools that support the definition of feature-based variability in model transformations. Transformations can be thus be selectively adapted.

In model transformations, a model is transformed into another model, the input model is typically left unchanged. In Figure 15 a model `M` is transformed into a model `K`. Both models are instances of different meta models. An important advantage of model transformations is that a clean separation between models and also between meta models can be achieved. Also, the different meta models can evolve independently.

There is also the notion of a model modification, where a model is modified 'in place', i.e. the input model is changed, and no additional output model is created. Since such a model modification is technically almost identical to a model transformation as defined above, this section focuses exclusively on model transformations.

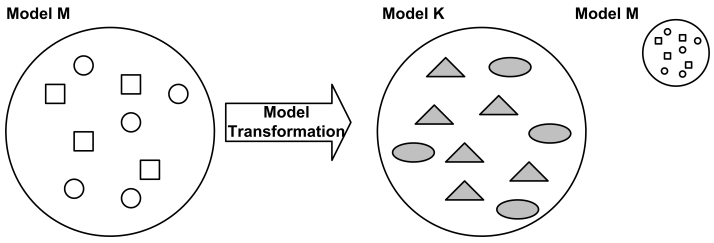

Fig. 15. Model transformations

As demonstrated in the previous section, both models and meta models can vary using the concepts and tools we developed. This directly leads us to the need of varying model transformations. New meta model elements brought by additional features must be added to the transformation workflow. Again, the adaptation of model transformations is only required in case the respective features are selected in the current configuration.

We solve this problem by applying AO to model transformations. The transformation language Xtend [39] has been extended with support for aspects.

Xtend supports the application of advices to model transformation functions. Only around advices are supported. Listing 10 shows the syntax of advices in Xtend. The keyword `around` is followed by a pointcut that selects the points where the advice should be applied. Any number of expressions can be executed

within the advice. By calling `proceed()`, the original transformation function is executed.

```
around [pointcut] :
   expression;
```

Listing 10. Around advices in Xtend

A pointcut consists of a fully qualified name and a list of parameter declarations. The asterisk character is used to specify wildcards. Listing 11 shows some examples of how pointcuts are specified. Parameter type polymorphism is considered when matching the pointcut.

```
my::Extension::definition      /* matches extensions with the
                                  specified name */
org::oaw::*                   /* matches extensions prefixed
                                  with 'org::oaw::' */
*Operation*                  /* matches extensions containing
                                  'Operation' */
*                           /* matches all extensions */
```

Listing 11. Pointcut specifications in Xtend

As we want to apply advices only in case a certain feature is selected in the current configuration, we need to link advices to features. Figure 16 illustrates how this is done. Transformation aspects are connected to features in the oAW workflow. The advice is then only applied to the transformation in case the respective feature is selected.

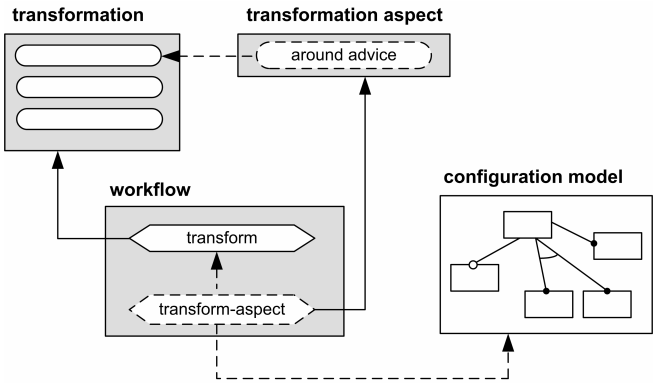

Fig. 16. Variability in model transformations

Imagine, one wants to deploy an optional tracing interceptor to a system. The runtime infrastructure (platform neutral component level) supports the use of interceptors for any component. Interceptors are available in libraries. If the model configures interceptors for a given component, the generated OSGi activator actually instantiates them, instantiates a proxy for each component and adds the interceptors to that proxy. Listing 12 shows the advice that is applied to the transformation function `transformPs2Cbd`.

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

Listing 12. Optional tracing interceptor advice

If the feature `Tracing` is selected, the transformation must make sure that the appropriate interceptor is configured for the components. This dependency is illustrated in the workflow shown in Listing 13.

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

Listing 13. Tracing workflow

Applying aspects to model transformations realizes positive variability on transformation level. As an alternative, one could also develop the overall transformation and exclude transformation steps in case a certain feature is selected. This can be realized by calling the `hasFeature(featureName)` function from within transformations and can be considered negative variability on transformation level.

4.1 Related Work

The approach presented in [37] includes an aspect-oriented extension to a model-to-text transformation language. Those so-called high-oder transformations can be used to represent variability in product line engineering. The approach is similar to our approach as an existing transformation language has been extended with support for aspects. An important difference is that our approach supports linking these aspects to features defined in a feature model. The approach in [37] does not include this capability.

5 Expressing Variability in Code Generation Templates

This section describes concepts and tools that support the definition of feature-based variability in code generation templates. Generators can be thus be selectively adapted.

A generator generates some textual output (e.g. code, build scripts, XML configuration files) from a model. The generator operates on the meta model of the DSL and thus has to know this meta model. Figure 17 illustrates the relationship between model, meta model, and generator.

A template language including a suitable template engine is used to generate the output. Templates support the generation of any text-based output.

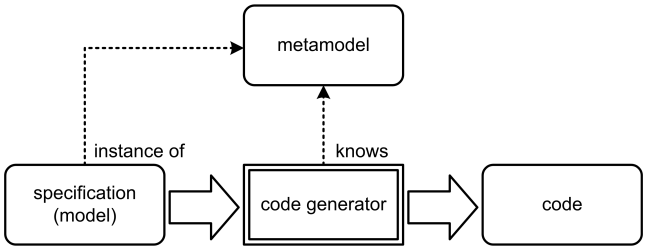

Fig. 17. Code generation

In our approach code is generated from the final solution space model. As meta model, models and the respective transformations can vary, we also need to incorporate variability into the code generation process. We use AO on code generation level for this purpose. The template language Xpand [39] provides support for template aspects.

Xpand supports around advices whereby any execution of a template definition can be a join point. The definition name part of a pointcut must match the fully qualified name of the join point's definition. The asterisk character is used to specify wildcards. Listing 14 shows the syntax of Xpand advices. The original template definition can be called with `proceed()`.

```
«AROUND qualifiedDefinitionName(parameterList) FOR type»
    a sequence of statements
«ENDAROUND»
```

Listing 14. Xpand around advice

The parameters of the definitions we want to add our advice to can also be specified in the pointcut. The rule is that the type of the specified parameter must be the same or a super type of the corresponding parameter type of the definition to be called (i.e. polymorphic dispatch is used on all arguments). Additionally one can set the wildcard at the end of the parameter list to specify that there might be none or more parameters of any kind. Listing 15 shows some examples.

```
my::Templ::def()          /* template definition without parameters */
my::Templ::def(String s) /* template definition with exactly one
                            parameter of type String */
my::Templ::def(String s,*) /* template definition with one or more
                              parameters where the first parameter
                              is of type String */
my::Templ::def(*)         /* template definition with any number of
                            parameters */
```

Listing 15. Wildcards in parameter definitions

Imagine, one wants to include a very simple tracing functionality to the components of the system. We advice the template that generates the code for the operations of the components and add a simple tracing statement at the end of the generated code. Listing 16 shows the respective template advice.

```
«AROUND templates::operation FOR Component»
```

```
  <<targetDef.proceed()>>
   System.out.println("simple tracing: within <<targetDef.name>>");
<<ENDAROUND>>
```

Listing 16. Xpand tracing aspect

Also, we want generator advices only to be applied in case a certain feature is selected in the current configuration. The dependency between template aspects and features is specified in the workflow (cf. Figure 18).

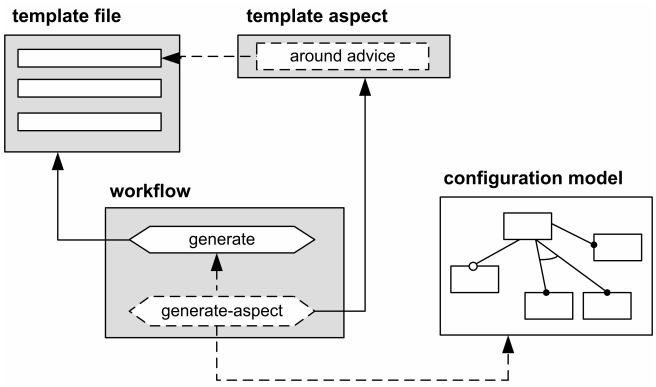

Fig. 18. Variability in code generation templates

Applying aspects to code generation templates realizes positive variability on generation level. Again, as an alternative, one could also develop the overall generator and exclude templates in case a certain feature is selected. This can be realized by calling the `hasFeature(featureName)` function from within templates and can be considered negative variability on generator level.

5.1 Related Work

In [32] a generative approach called Framed Aspects is proposed. Framed Aspects combine AOP with frame technology to modularize crosscutting feature implementations and improve evolution of product lines. In contrast, our approach includes aspects directly into code generation templates to incorporate features into the code generation process. The approach presented in [32] only parameterizes aspects.

The transformation engine of ArcStyler, CARAT [26] supports 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. However, our approach is more generic because it also supports quantification — the ability to select a set of join points using a pointcut to add generator behavior in several places at once.

6 Expressing Variability in Code

For some features developing a custom generator might be too costly. Thus, for economic reasons, certain parts of the product will still be implemented manually. In this case variability has to be considered at code level.

We use AO on code level to realize positive variability. AspectJ [28] and CaesarJ [4] are popular candidates for implementing features on code level.

Sometimes a feature might be requested for which the product line architecture may not provide the required configuration or customization hooks. Either one has to manually tweak the generated code to accommodate the variant, or the product line architecture has to be adapted to include the additional hooks. The latter approach is desirable but for reasons of versioning, coordination, or time pressure it is often not realistic. AOP can be very useful in this case. One can hook into generated (or manually written) code at places where the product line architecture does not provide hooks. Thus the necessary change can be accommodated without changing the product line architecture and without manually changing the generated code — the change is external in the aspect.

To support negative variability on code level we extended XVar [39] [22] with additional capabilities. It is possible to include special comments to the code that define the dependencies to features. In case the respective feature is not part of the current configuration, the code implementing this feature is removed from the code base. Listing 17 shows an example. A light switch can switch both normal lights and dimmable lights. The feature `Dimmable Lights` is optional. The example illustrates how special comments (starting with #) specify the dependency to the feature. In case the feature is selected the first statement is part of the code base, in the other case the second statement is included.

```
public void execute() {
    ...
    //# dimmableLights
    parseLightsToSwitch(changedLights, status.getStatusLightLevel());
    //~# dimmableLights

    //# !dimmableLights
    parseLightsToSwitch(changedLights);
    //~# !dimmableLights
}
```

Listing 17. Negative variability on code level

6.1 Related Work

The main focus of research on AOSD in SPLE is targeted towards variability implementation using AOP languages.

In [34] it is demonstrated how CaesarJ helps to overcome the deficiencies of feature-oriented programming (FOP) and AOP for implementing variability. CaesarJ supports both multi-abstraction modules and join point interception.

The approach described in [3] introduces Aspectual Mixin Layers (AML) which integrate both AOP and FOP by introducing aspects into mixin layers and providing aspect refinement.

In [1] the capabilities of AO to implement variability are evaluated according to defined criteria. The evaluation has shown that AOP is especially suitable for variability across several components, i.e. crosscutting variability.

In [7] it was demonstrated how AspectC++ can be employed in a weather station product line. The use of AspectC++ simplified the development process since crosscutting product line features could be directly mapped to aspects. Aspect C++ enabled configuring the appropriate level and therefore enhanced the code reusability.

7 Home Automation Case Study

The case study to illustrate our approach is a home automation system (see also [41]), 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 SmartHome 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 cost savings drive the need for a Smart Home product line and are the main causes of variability.

The remainder of this section will explain how the techniques introduced in Section 2 were used to implement the Smart Home product line.

7.1 Problem Space Modeling

In the problem space, Smart Home systems are formally described. A problem space meta model is defined that contains entities such as buildings, floors, rooms, the various kinds of sensors, and actuators. Note that this model does not contain anything concerned with software or computing hardware. It formally describes domain requirements. Figure 19 shows parts of the problem space meta model of Smart Home. Buildings contain floors and floors contain rooms. Staircases connect the different floors in the house. Different kinds of devices are located in rooms.

Using this meta model, a DSL is built which supports modeling Smart Home systems from the perspective of a building architect or a home owner. The syntax is semi-graphical and a customized tree view is developed. Figure 20 shows an

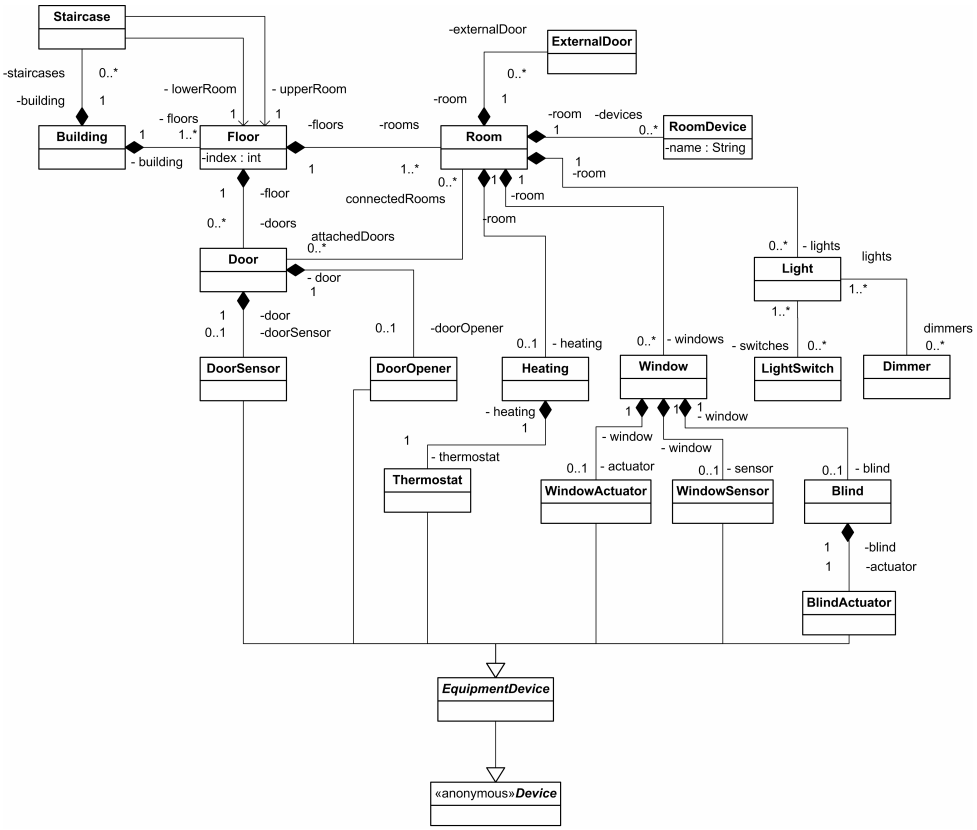

Fig. 19. Problem space meta model of Smart Home

example of a house that is modeled using the DSL. This DSL is a creative construction DSL. Note that in a real-world system, the building data would be extracted from a CAD system.

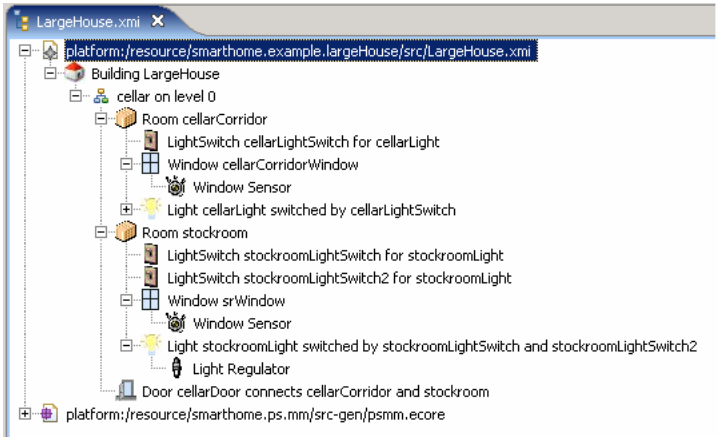

Fig. 20. Example house

7.2 Solution Space Modeling

The solution space comprises a component-based architecture. Figure 21 shows the types viewpoint this component meta model. Additional viewpoints are defined to express component instances, their connections, hardware structure, as well as the mapping of software component instances onto hardware nodes. There is no need to provide a sophisticated concrete syntax for that domain, since the models are created by model transformations from problem space models. Note that this meta model is platform independent in that the generic component architecture can be mapped onto various target platforms such as OSGi [40], Spring [43] or JEE [45]. The case study implementation uses OSGi.

To a large extent, Smart Home systems consist of a specific arrangement of pre-built sensors and actuators (although a specific system can have custom devices). It therefore makes sense to keep a library of software components that control certain types of hardware. We use a combination of manually written code (i.e. the business logic) and models to represent these components in order to be able to use them to define a given system. Based on the problem space model, the transformation instantiates, wires, and deploys those library software components in the context of the software system generated for a particular building.

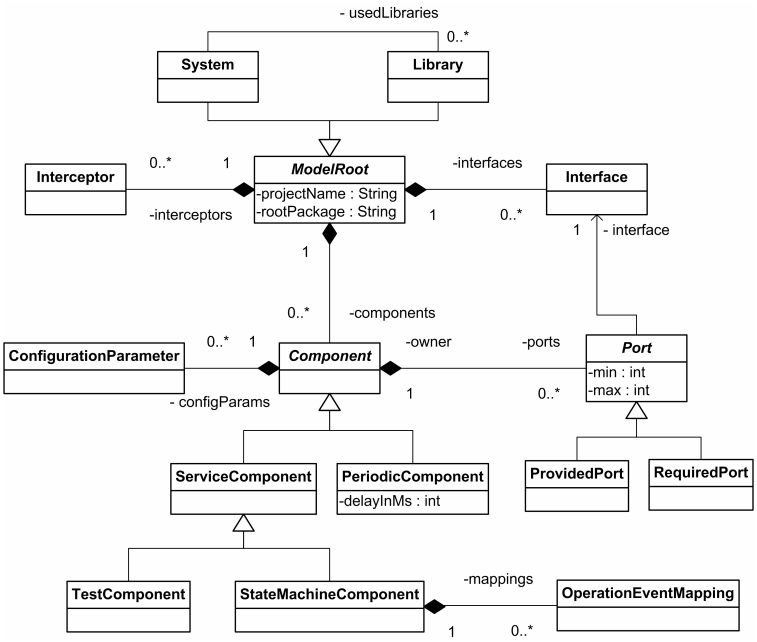

Fig. 21. Platform independent component meta model: Types viewpoint

7.3 Solution Space Implementation

In realistic Smart Home scenarios, there are several target implementation technologies. For example, computing platforms and networking/bus technologies change depending on the level of sophistication of a product. In our case study we selected OSGi as a target platform. We defined an extended OSGi meta model (cf. Figure 22) and mapped the platform independent component meta model to the OSGi meta model using a model-to-model transformation.

The formally defined mappings allow us to automatically transform a problem space model (cf. Figure 20) into a platform independent component model and into an OSGi model, in turn. Finally, we generate code from the OSGi model.

Figure 23 provides an overview of the meta models, models, and transformations used in the SmartHome case study. Meta model parts common to the component meta model and the OSGi meta model were modeled in separate meta models. Common parts include data types and operations. We developed a tool called XJoin [4] that supports adding relationships between meta models to join them. The separate meta models keep their own identities and the partial models do not need to know about the other ones.

[4] XJoin is part of openArchitectureWare 4.2 and can be downloaded from http://www.openarchitectureware.org/

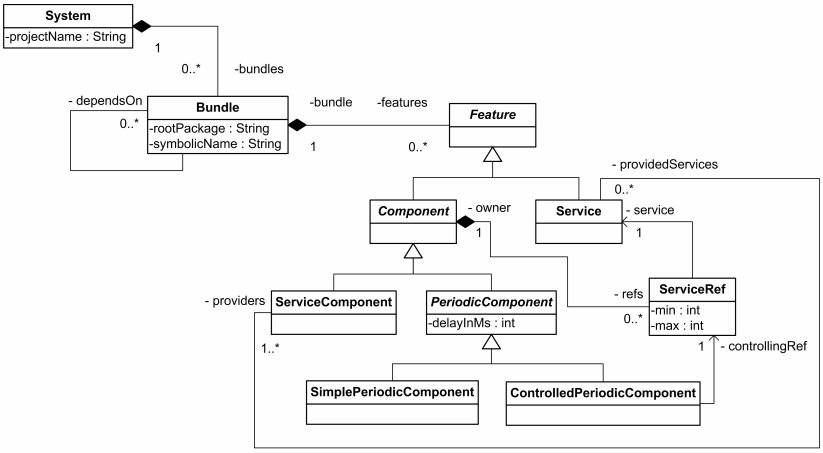

Fig. 22. OSGi meta model

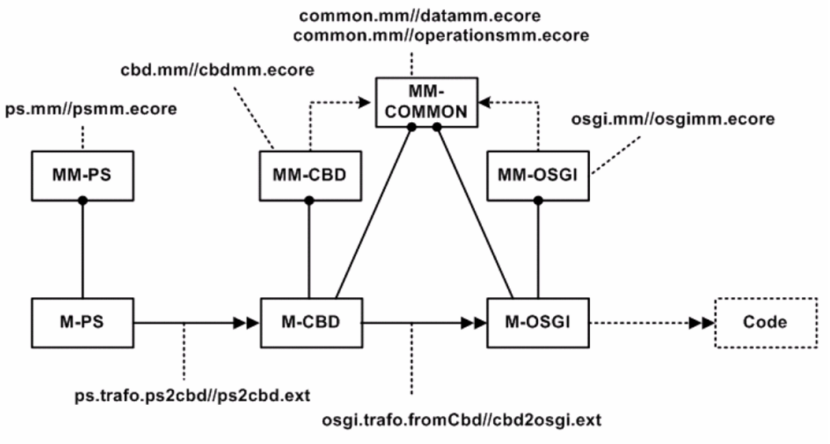

Fig. 23. Smart Home models and transformations overview

7.4 Orthogonal Variability

We use a global feature model to incorporate features that are orthogonal to the house structure. Those features require to configure the entities of the house, the transformation and generation process in one way or the other. For implementing this orthogonal variability we use the concepts and tools described in Section 2 and subsequent sections. We use pure::variants [42] for feature modeling.

Figure 24 shows parts of the feature model of Smart Home. It provides several automation features that let the house act autonomously according to defined policies. Additionally it provides debugging features and several alternatives with respect to deployment.

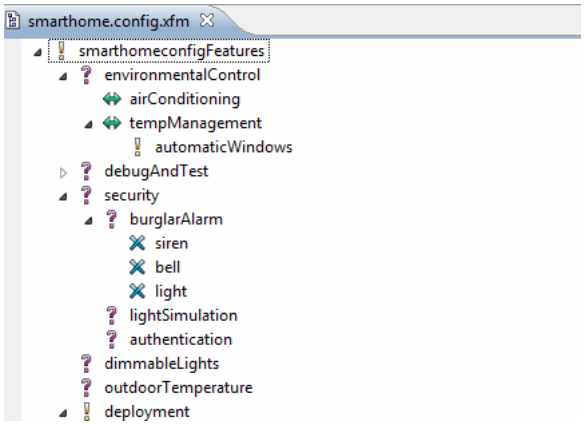

Fig. 24. Smart Home feature model

Due to space limitations we can only provide details for two of these features, namely the automation feature *Automatic Windows* and the debugging feature *Reflective Data Structures*.

Automatic Windows The Automatic Windows feature automatically opens the windows if the temperature is a room is above a certain threshold and closes them again if the temperature is below a certain threshold. In order for this feature to be included in a configuration, the necessary devices have to be woven into the building model. Figure 25 shows the aspect model that is responsible for weaving the respective features into the example house shown in Figure 20. The pointcut expressions used in the aspect model are shown in Listing 18.

```
rooms(Building this) : floors.rooms.select(e|e.windows.size>0);
windows(Building this) : rooms().windows;
thermoName(Thermometer this) : ((Room)eContainer.name.toFirstLower() +
                                "Thermometer";
```

Listing 18. Pointcut expressions

`rooms` returns all rooms that have windows. To all of them a thermometer should be added and `thermoName` is a helper function that creates a sensible name for this thermometer. `windows` returns all windows of these rooms and a window actuator is added to them.

The resulting (woven) model has a thermometer in each room to measure the current temperature and a window actuator for each window to be able to automatically open the windows. If the Automatic Windows feature is present in the configuration model, XWeave weaves the content of the aspect model into the house model. This dependency is specified in the workflow.

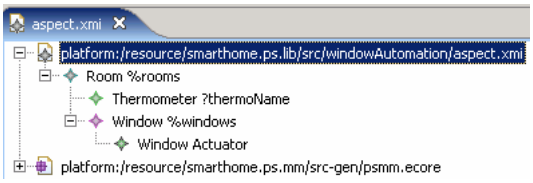

Fig. 25. Window automation aspect

The additional devices required for the Automatic Windows feature are automatically transformed and the respective code is generated. There is no need to change the transformations or the generators as the variability is purely handled on model level.

The business logic of the Automatic Windows feature (the actual opening of the windows in case the temperature reaches a certain threshold) is pre-built in the form of a library component and included into the final product in case the feature is selected. Since all pre-built components have representations on model level, we can process them using model transformations. Based on the information in the models it is possible to determine whether a given component is part of a product or not.

Reflective Data Structures 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. For this to work certain parts of the system need to include a specific reflection layer that is used by that GUI. Specifically, if component instance states should be inspected, the data structures representing the state need to be "reflective" and 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 the Debug feature is selected or not.

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. Listing 19 shows the templates that will be adviced by the template aspect.

```
«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»
```

Listing 19. Templates to be adviced

The piece of Xpand code in Listing 20 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»
```

Listing 20. Template aspect

Of course, to make this work as desired, we have to couple the aspect to the configuration model. This dependency is again specified in the workflow.

8 Summary and Future Work

8.1 Research Questions Revisited

In Section 1 we define the research question we aim to answer based on data from the case study we conducted at Siemens AG.

This research question is:

Where and how can software product line development benefit from AOSD, MDSD, and their combination?

We started by modeling the problem space, i.e. by creating a meta model of the domain requirements. Next, we defined a meta model of a platform neutral component-based architecture and mapped the domain entities to components. We then defined a meta model of OSGi, created the mapping from the component-based architecture meta model and developed the code generator.

The meta models, mappings, and generator already provided the appropriate means for dealing with all variability concerning the structure of the house. Developers are now able to model different instances of houses and automatically generate the appropriate OSGi code from them.

This leads us to the first important benefit: By integrating MDSD into SPLE, creative construction DSLs can handle variability with respect to the structure or behavior the DSL intends to model. A DSL allows to create models that are instances of a defined meta model, which is the meta model of the domain. Variability with respect to domain entities and their relations can be efficiently handled using a MDSD approach. Although, DSLs only deal with structural variability and in product lines configurative variability is an important issue as well.

In our case study we used AO concepts and techniques on model, transformator, generator, and code level to handle *orthogonal variability*. By orthogonal we mean variability that affects multiple domain entities and their subsequent processing (i.e. transformation) steps. We used a feature model to describe the configurative variability of the home automation system and realized the features using AO at the appropriate levels.

This leads us to the second important benefit: Variability that is orthogonal (and usually configurative) to the structure the DSL intends to model can be efficiently addressed using AO techniques. Features can thus be localized in aspects and combined with the core as desired. The higher the abstraction level is, the fewer variation points exist. Features expressed on models level are thus inherently simpler than features expressed on code generation level. We therefore argue to always express features on the highest possible level.

In AO systems, whenever more than one aspect is applied to a system, aspect precedence is an issue, i.e. the advice ordering in case more than one advice is applied at the same join point. We solve this by explicitly coding the order in which aspects are applied in the workflow. Workflow steps are executed in sequential order which means that the weaving process is strictly sequential. One aspect is applied after the other in the order specified in the workflow. In the case of XWeave woven models can be the input of a weaving process again as XWeave does not distinguish between a woven and a non-woven model. The fact that one can actually look at the woven model has a positive impact on understandability and usability.

Another important contribution of our approach is the integration of a variant management tool (in this case pure::variants) into the MDSD tool chain. Within

our tooling, configuration models can be queried and hence, activities can be performed based on the presence or absence of features in a configuration model.

8.2 Future Research

In the future we will work on the improvement of the tooling we introduced in this article. We will work on a better visualization of the dependencies between MDSD artifacts and features. For example, the workflow components, transformation steps, and templates that depend on a feature can be highlighted in the same color. This facilitates maintainability of features.

Also, XWeave currently only supports additive weaving. In the future we plan to extend XWeave in order to support changing or overriding existing base model elements using aspects. Another possible extension of XWeave is support for symmetric model weaving. This kind of weaving does not distinguish between aspect and base models. Models are woven together according to defined rules to form the final system. Our observations show that in SPLE, typically, a core capturing the commonality within a portfolio of similar products can be identified. Features are increments in functionality and thus match well with the idea of asymmetric weaving.

Additionally we will apply our approach in a second industrial case study to further validate the developed concepts and tools.

Acknowledgments. This work is supported by AMPLE Grant IST-033710. The authors would like to thank Christa Schwanninger for her valuable comments on earlier drafts of this article.

References

[1] Anastasopoulos, M., and Muthig, D. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR* (2004), pp. 141–156.

[2] AOSD Community Wiki. Glossary. http://www.aosd.net/wiki/, 2007.

[3] Apel, S., Leich, T., and Saake, G. Aspectual mixin layers: aspects and features in concert. In *ICSE* (2006), pp. 122–131.

[4] Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. An overview of CaesarJ. *Transactions on AOSD I, LNCS 3880* (2006), 135 – 173.

[5] Baniassad, E. L. A., and Clarke, S. Theme: An approach for aspect-oriented analysis and design. In *ICSE* (2004), pp. 158–167.

[6] Batory, D., Sarvela, J. N., and Rauschmayer, A. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 187–197.

[7] Beuche, D., and Spinczyk, O. Variant management for embedded software product lines with pure: : consul and aspectc++. In *OOPSLA Companion* (2003), pp. 108–109.

[8] BigLever. Gears. http://www.biglever.com/, 2007.

[9] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M. P., Bakker, J., Tekinerdogan, B., Clarke, S., and Jackson, A. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Tech. Rep. AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University, 18 May 2005 2005.

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

[11] Coplien, J. O. *Multiparadigm Design for C++*. Addison Wesley, 1998.

[12] Cottenier, T., van den Berg, A., and Elrad, T. Joinpoint inference from behavioral specification to implementation. In *ECOOP* (2007), pp. 476–500.

[13] Czarnecki, K., and Antkiewicz, M. Mapping features to models: A template approach based on superimposed variants. In *GPCE* (2005), pp. 422–437.

[14] Eclipse. Eclipse modeling framework (EMF). http://www.eclipse.org/emf, 2007.

[15] Eclipse. Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/, 2007.

[16] eCos. Operating system. http://ecos.sourceware.org/, 2007.

[17] Fabro, M. D. D., and Valduriez, P. Semi-automatic model integration using matching transformations and weaving models. In *SAC* (2007), pp. 963–970.

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

[19] Filman, R. E., and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. Tech. rep., 2000.

[20] Gray, J., Lin, Y., and Zhang, J. Automating change evolution in model-driven engineering. *IEEE Computer 39*, 2 (2006), 51–58.

[21] Greenfield, J., Short, K., Cook, S., and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[22] Groher, I., and Voelter, M. Expressing feature-based variability in structural models. In *Workshop on Managing Variability for Software Product Lines* (2007).

[23] Groher, I., and 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, 2007), ACM Press, pp. 35–40.

[24] Heidenreich, F., and Wende, C. Bridging the gap between features and models. In *Second Workshop on Aspect-Oriented Product Line Engineering* (2007).

[25] IBM. Rational Rose. http://www-128.ibm.com/developerworks/rational/, 2007.

[26] Interactive Objects. Arcstyler. http://www.interactive-objects.com/, 2007.

[27] Kannan, M., and Ramesh, B. Managing variability with traceability in product and service families. In *HICSS* (2002), p. 76.

[28] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ. In *ECOOP* (2001), pp. 327–353.

[29] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP* (1997), pp. 220–242.

[30] Kruchten, P. The 4+1 view model of architecture. *IEEE Software 12*, 6 (1995), 42–50.

[31] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schröder-Preikschat, W. A quantitative analysis of aspects in the ecos kernel. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), ACM, pp. 191–204.

[32] Loughran, N., and Rashid, A. Framed aspects: Supporting variability and configurability for aop. In *ICSR* (2004), pp. 127–140.

[33] Maeder, P., Riebisch, M., and Philippow, I. Traceability for managing evolutionary change. In *SEDE* (2006), pp. 1–8.
[34] Mezini, M., and Ostermann, K. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE* (2004), pp. 127–136.
[35] No Magic. Magicdraw. http://www.magicdraw.com/, 2007.
[36] Object Management Group. XML Metadata Interchange (XMI) specification. http://www.omg.org/mof/, 2007.
[37] Oldevik, J., and Haugen, O. Higher-order transformations for product lines. In *SPLC* (2007), pp. 243–254.
[38] openArchitectureWare. http://www.openarchitectureware.org, 2007.
[39] openArchitectureWare. User guide version 4.2. http://www.eclipse.org/gmt/oaw/doc/4.2/openArchitectureWare-42-reference.pdf, 2007.
[40] OSGi Alliance. Osgi framework. http://osgi.org/, 2007.
[41] Pohl, K., Böckle, G., and van der Linden, F. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, Berlin, 2005.
[42] pure systems. pure::variants. http://www.pure-systems.com/, 2007.
[43] Spring Framework. http://www.springframework.org/, 2007.
[44] Stahl, T., and Völter, M. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley & Sons, 2006.
[45] Sun Microsystems. Java Enterprise Edition. http://java.sun.com/javaee/, 2007.
[46] Telelogic. DOORS. http://www.telelogic.com/products/doors, 2007.
[47] Voelter, M., and Groher, I. Handling variability in model transformations and generators. In *7th OOPSLA Workshop on Domain-Specific Modeling* (2007).
[48] Voelter, M., and Groher, I. Product line implementation using aspect-oriented and model-driven software development. In *SPLC* (2007), pp. 233–242.