Multi-Level Product Line Customization

Christoph ELSNER [a], Christa SCHWANNINGER [a],
Wolgang SCHRÖDER-PREIKSCHAT [b] and Daniel LOHMANN [b]
[a] *Siemens Corporate Technology*
Software & Engineering 2, Erlangen, Germany
{christoph.elsner.ext,christine.schwanninger}@siemens.com
[b] *Friedrich-Alexander University Erlangen-Nuremberg*
Department of Computer Science 4, Erlangen, Germany
{wosch,lohmann}@cs.fau.de

Abstract. Managing and developing a set of software products jointly using a software product line approach has achieved significant productivity and quality gain in the last decade. More and more, product lines now are becoming themselves entities that are sold and bought in the software supply chain. Customers build more specialized product lines on top of them or derive themselves the concrete products. As customers have different requirements, whole product lines now may vary depending on customer needs—they need to be *customized*. Current approaches going beyond the scope of one product line do not provide appropriate means for customization. They either are tailored to specific implementation techniques, only regard customization on few levels (e.g., only source code level), or imply a lot of manual effort for performing the customization.

The PLiC Approach tackles this challenge by providing a generic, reusable reference architecture and methodology for implementing such customizable product lines. In the reference architecture, a product line consists of so-called product line components (PLiCs), which are flexibly recombinable slices of a formerly monolithic product line, thereby maintaining strict separation of concerns. The approach furthermore comprises a tool-supported methodology for recombination of PLiCs based on customer needs and thus minimizes manual intervention when customizing. We implemented the PLiC Approach for a complex model-driven product line, where it facilitates comprehensive customization on various levels in the models, the model transformation chain, and in the source code with reasonable effort. This gives evidence that our approach can be applied in various other contexts where the same or fewer customization levels need to be considered.

Keywords. Software Product Line Development, Model-Driven Development, Software Product Line Architecture

1. Introduction

Product line engineering [7,25,27] denotes a collection of engineering techniques supporting the efficient reuse of a common set of core assets when developing similar products. There exist different maturity stages for implementing a software product line [8]. The more mature stages imply that the process of deriving a product can vastly be automated [12]: A product is specified (i.e., configured) in a *problem space*, where domain-specific terms and concepts exists that the customer is familiar with. An automated *map-*

ping of this specification to appropriate implementation artifacts in the *solution space* then generates the concrete product. For example, a model-driven product line for building automation uses models as input that specify a house and its interior devices in problem space. Automated model transformations and code generation then map these models to house-specific source code, which, together with manually implemented source code, constitutes the solution space.

This automated type of product line engineering, which minimizes the implementation knowledge for product derivation, is to prefer even more when customers acquire a whole product line to derive the products themselves, or when they reuse it to build more specialized product lines. The scenario that a whole product line itself is the unit of sales is also called software supply chain [19] or software ecosystem [10] in recent work. Given this as a prerequisite, the scope of a product line in the supply chain should be adapted to the needs of the customer prior to sales. We will denote this case as *product line customization*. Doing so, different market segments can be satisfied more appropriately, leading to better market coverage and finally to more sold product line instances. A further consequence is, that product line customization on an automated product line needs to be performed in the problem space, in the solution space, and in the mapping between them.

For example, commercial embedded operating systems such as VxWorks [37], Windows CE [38], and ProOSEK [28] are highly configurable and can be regarded as automated product lines. A customer acquires one of these product lines and uses its configuration interface to derive a concrete product tailored to his needs. However, the product lines vary themselves depending on customer needs. The Windows CE development environment, for example, exists in special versions for PocketPC, PocketPC Phone Edition, Smartphone, and industrial applications, each allowing to derive a variety of operating systems to meet the needs of the developers. The situation is similar for VxWorks from Wind River, which furthermore offers symmetrical multiprocessing capabilities – a highly crosscutting feature – as optional add-on. The ProOSEK product lines, finally, differ in the embedded hardware they support.

There are several implementation approaches that go beyond the scope of one single product line. They do not explicitly consider product line customization, as characterized above, and have further drawbacks: They are either tailored to specific implementation technologies, they confine themselves to a restricted set of customization mechanisms, or lack concepts for automating the process of product line customization and product derivation (see related work, Section 7). However, all these points need to be addressed when developing a concept for product line customization, this is, when the product line itself becomes a product which needs to be tailored.

In one of our industry cooperations, we explicitly met the case of product line customization. A set of similar embedded product lines has been implemented by using a common code repository. A script file selects the corresponding files for a concrete product line, which is the commodity sold to the customer. It also initiates the interpretation of preprocessor directives for those files in which concerns related to different product lines are intermingled. According to our experience, such code-centric methods can be regarded as state-of-the-art in industry. Due to a non-disclosure agreement, further details cannot be published. However, already from the general description of the process, we can derive the following shortcomings:

Intermingled Concerns Within the Asset Base
In the above described example all files are managed in the same repository and scripts decide if they are included into a specific product line or not. To be able to relate a file to the product line(s) it belongs to, strict guidelines or methods are required. Implementing variability with preprocessor directives results in intermingled concerns within a file. It is hard to decide which part of which file will actually be included into which product line and, finally, in which product. Furthermore, preprocessor directives typically render advanced editor support unusable, for example when a function or variable is declared in both the *if*- and *else*-branch of a preprocessor statement. Depending on the number of directives, they strongly impair readability and comprehensibility of source artifacts. [30]

Intermingled Concerns within the Monolithic Product Line Generator
The knowledge on how to produce a certain product line is encoded implicitly in the script file and controlled by various script parameters. It must perform several tasks, for example file selection, preprocessing, as well as generating the problem space representation (e.g., a feature model or a configuration interface) and the derivation infrastructure of the target product line. The multitude of tasks and the monolithic nature of the script file make it very hard to understand, maintain, and adapt.

To approach these two problems of intermingled concerns, both within the asset base and the generator, we regard the *separation of concerns* (SoC) as a key design principle. This means that optional or alternative parts of a product line (i.e., the customizable parts) reside in a dedicated artifact. We call such a part of a product line a *product line component (PLiC)*. It constitutes a slice of a product line that comprises the problem space, mapping, and solution space artifacts required for customizing the product line. The product line generator than composes the PLiCs according to customer requirements for creating a customized product line.

To tackle product line customization, we developed the PLiC Approach. It provides a generic, reusable reference architecture and a methodology for implementing customizable product lines. In the reference architecture, a product line consists of so-called product line components, which are flexibly recombinable slices of a formerly monolithic product line, thereby maintaining strict separation of concerns. The approach comprises a tool-supported, automated methodology for recombining PLiCs based on customer needs and thus minimizes manual intervention when customizing. We implemented the PLiC Approach for a model-driven product line, where it facilitates comprehensive customization on various levels in the models, the model transformation chain, and in the source code. In particular, we make the following contributions:

- We present a reference architecture for customizing product lines, the PLiC Reference Architecture, which is tailorable to arbitrary product line customization settings (Section 2).
- We discuss possible implementation alternatives for product line customization, thereby showing that SoC can be achieved regardless of the implementation technologies used (Section 3).
- We present a tool-supported methodology for composing the PLiCs, which minimizes the manual intervention during customized product line creation (Section 4).

We illustrate the PLiC Approach with a case study of a small product line customization. We customize the SmartHome [36,14] product line with a PLiC for safety augmenting it

with capabilities for triple modular redundancy and error injection. Each of the Sections 2 to 4 therefore first describes an example problem, then presents the general solution to solve the problem, and finally maps this solution to the example case. We will present further results of our case study implementation in Section 5. In Section 6 we discuss them with respect to generalizability and scalability of the approach, and, finally, we address related work and draw a conclusion in Sections 7 and 8.

2. A Reference Architecture for Customizing Product Lines

In this section, we present our generic reference architecture for developing customizable product lines, the PLiC Reference Architecture. It is tailorable to a wide range of settings. For illustration, we apply it to the model-driven product line SmartHome.

2.1. Example Problem

SmartHome [36,14] is a product line developed for construction experts such as architects and interior designers. It has been created in the context of the research project AMPLE [1] and is based on the widely-used model-driven framework openArchitectureWare (oAW) [26], which in turn is based on Eclipse. SmartHome facilitates modeling of buildings and their electrical interior devices in problem space and generates the software from the model for automatically controlling these devices.

Originally developed for standard homes, the product line now shall be extended to serve the market for industrial buildings as well. To segment the market, the assumed business model of our case study is to offer a standard version of the product line *as well as* an extended version, which includes advanced safety features.

2.2. Product Line Components

A *product line component (PLiC)* is a slice of a product line that encapsulates optional or alternative parts of the product line. It comprises the problem space configuration, the mapping (e.g., via file selection or code generation), and the solution space artifacts required for customizing the product line for a specific customer group. The notion of component in the term PLiCs stems from the fact that, on the one hand, they can be plugged into product lines to extend them and, on the other hand, points at the conventions they have to follow to do so.

For the rest of the paper, we will generally refer to a PLiC as an additional customization, whereas we will call the entity to be customized the *base product line*. This implies a hierarchical product line approach [9] and matches our case study domain where the Safety PLiC customizes the SmartHome base product line. In ideal case, also the base product line is a regular PLiC. However, when dealing with a legacy product line, it may also be treated as a special entity to avoid refactoring.

2.3. PLiC Reference Architecture

The *PLiC Reference Architecture* provides a framework for integrating PLiCs into a base product line. It must be generic and consider SoC as a key architectural principle. These constraints can be mapped to the following *architecture-relevant requirements*:

1. Support for Arbitrary Product Lines
   The PLiC Reference Architecture must not depend on a specific product line architecture. Thus, it has to support arbitrary product lines given a certain predetermined infrastructure technology (e.g., EMF [31], feature modeling [6]) and build tooling (e.g., make, Ant).
2. Separating Concerns of Base Product Line and PLiCs
   All artifacts of PLiCs and the base product line should remain clearly separated and interact with each other in a well-defined way (see Section 1, shortcoming 1).

Based on these requirements we developed the PLiC Reference Architecture as shown in Figure 1. The strategy is to implement as much as possible in the lower layers of the hierarchy to foster reusability. Whereas layer 1 provides support for arbitrary product lines, layers 2 and 3 consider SoC as a key architectural principle:

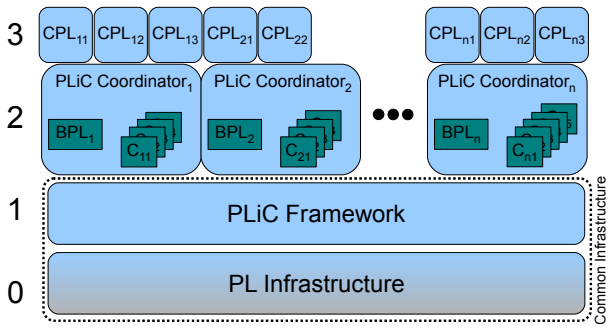

Figure 1. The PLiC Reference Architecture

Layer 0. The architecture is based on a common *product line infrastructure* like commercial product line tooling [6] or model-driven product-line-enabled frameworks [26].
Layer 1. Based on Layer 0, the *PLiC Framework* provides base functions for defining and coordinating the overall product generation process. These functions are the building blocks for customization. They are used by the above layer to easily specify the points during product generation where PLiCs can customize the base product line (details will follow in Section 3). The PLiC Framework supports arbitrary product lines, as it does not imply the use of specific types of configuration models or assume predefined steps of product generation.
Layer 2. For each base product line (BPL), a *PLiC Coordinator* needs to be designed. It specifies the concrete customization points (the customization *interface*) at which a PLiC can extend the BPL. During product generation, the PLiC Coordinator ensures that PLiCs may extend a BPL only at these predefined points. As each PLiC itself contains all information necessary to generate its part of the product, there does not exist a monolithic product generation facility as in repository-based approaches (Section 1, shortcoming 2).
Layer 3. *Customized product lines* (*CPL*s) are specified on top of a PLiC Coordinator. A CPL represents a BPL enriched (in problem space, mapping, and solution space) with PLiCs. Following the principle of SoC, we avoid physical copying and merging of BPL and PLiC files. Instead, a CPL is a simple placeholder that forwards the actual product generation to the PLiC Coordinator, which dynamically composes the product generation

logic of the base product line and the chosen PLiCs. [1] A complete customized product line delivered to a customer therefore comprises the PLiC Coordinator, the BPL, the set of chosen PLiCs, and the CPL, which is the frontend for the customer.

2.4. Example Solution

In our example scenario, the SmartHome product line is extended with a *product line component for safety*, which can be employed to meet the requirements of industrial building automation. The PLiC Reference Architecture model must be applied accordingly. The PLiC Framework has to be developed to extend the model-driven product line infrastructure oAW to provide a general, easy-to-use interface for all artifacts that shall be customizable, particularly, models, metamodels, transformators, generators, and code (Section 3). Furthermore, a PLiC Coordinator for the specific product generation process of SmartHome and the customization needs of the Safety PLiC must be developed (Section 4) and, finally, SmartHome must be adapted to this PLiC Coordinator (Section 5).

3. Maintaining Separation of Concerns When Implementing Customizations

After applying the PLiC Reference Architecture to a product line, it is still open how to actually implement the customizations that a PLiC shall apply to the base product line. In the following, we will argue that any artifact involved in product generation can be customized, whereas strict separation between base functionality and additional, customized functionality can be achieved.

3.1. Example Problem

The SmartHome product line facilitates modeling of buildings and their electrical devices and generates the specific software for all device controllers. Code generation builds on top of a multi-level model-driven process involving several technologies, in particular models, metamodels, model transformations, code generation, and manually implemented application logic components. A PLiC may extend the base product line on all these levels for implementing a customization, so all these implementation technologies need to support SoC.

3.2. Implementation of Customizations

We divide the asset base of a product line into fundamentally three levels. In each level, customization needs may arise. Bottom-up, these levels are the passive artifact level, the active artifact level, and the top artifact level.

The *passive artifact level* consists of artifacts that either immediately contribute to end products (e.g., compiled code, configuration files) or that are transformed in some way prior to their contribution (e.g. models, source code). Passive artifacts may reside both within problem space (i. e., models) and solution space (all kinds of artifacts). The

[1] Keeping the BPL and PLiC files separate and avoiding copying has, besides a clear structure, further advantages. It hinders that differing versions of BPLs and PLiCs are used in different CPLs, as both multiple version keeping and reconciliation is very costly and should be avoided.

active artifact level contains those artifacts that transform other (usually passive) artifacts. They are, for example, compilers, model-to-model transformators, and model-to-code generators. This level also contains the logic to select passive artifacts according to the concrete product configuration. Therefore, it represents the mapping from problem space to solution space artifacts as well as the mapping between solution space artifacts. The *top artifact level* serves for specifying the overall product generation process and therefore the sequence for applying active artifacts.[2] It may consist of a hierarchy of make files or a script file describing the model transformation sequence. The artifacts of the top level together with the active artifacts are sometimes also referred to as product generation facility.

Separating concerns between the base product line and the PLiCs depends on the capabilities of the employed implementation languages and the level of invasiveness of the PLiC's intended customizations on each level. We distinguish three *classes of customizations*: noninvasively-implementable customizations, invasively-implementable customizations, and customizations by shifting to a higher level.

Noninvasively-implementable customizations do not require any changes to the base artifact. In the simplest case, the extension of a passive artifact is well-modularizable, can be implemented in a dedicated file, and can be composed by general-purpose compilers, weavers, linkers, etc. Active artifacts (e. g., model transformations), on the other hand, are often executed sequentially and therefore can be implemented separately as well. If the employed languages have advanced capabilities for SoC (e.g., aspect orientation [23, 22]), even fine-grained customizations can be implemented in the PLiC without need for adapting the base product line. Furthermore, current component frameworks such as Spring provide means for advanced SoC in common third generation languages by using interceptors.

Invasively-implementable customizations exploit the fact that refactoring of the base artifact often provides a viable solution for SoC. Various loose-coupling design patterns [17] (Observer, Mediator, etc.) can be applied for this purpose. Single program statements can also be factored out into dedicated methods to make them advisable by aspects, or to simply override them using inheritance.

In case the base product line consists of passive artifacts lacking mature mechanisms for SoC (e.g., XML, C) and performance constraints impede the use of certain design patterns, but there is still the need for fine-grained customizations, we advocate for solving the problem via *shifting to a higher level*. Then, the mentioned noninvasive and invasive customizations can be applied on the active instead of the passive artifact level. For example, it is possible to map arbitrary *textual* customizations to a customization problem on *text generation* level (e. g., JET, XPand), where loose-coupling design patterns may be applied. Furthermore, concerns can be separated in several domain-specific models and can be recombined either with custom-made model transformators or with the help of a model merging tool.

3.3. Example Solution

In the following, we will apply the strategy for implementing customizations to our safety extension for the SmartHome product line. Its model-driven product generation process

[2] Strictly seen, the active artifact level would also comprise the top artifact level. However, as we need to distinguish the two levels for the following classification, we keep both levels separate.

is based primarily on domain-specific EMF (Eclipse Modeling Framework) models. A building is described in a problem space model and automatically and repeatedly transformed first to a platform-independent and then to a platform-specific solution space component model. Finally, application glue code is generated. Business logic is implemented manually in Java. We will first give an overview of the various customization technologies used and then give some concrete examples.

3.3.1. Overview

Within the SmartHome customization case study, separation of concerns on various levels became necessary, both for *passive artifacts* such as models, metamodels, and source code and for *active artifacts* such as model transformators and model generators. All technologies and tools we use are publicly available as part of the Eclipse framework.

SoC for Passive Artifacts
On *source code level*, we use plain Java to implement well-modularized components and AspectJ [23,22] for crosscutting concerns. To merge concerns on *model and metamodel level*, we use the aspect model weaver *XWeave* [18] for basic model composition tasks. We implement more advanced model weaving logic by shifting the problem to the active artifact level. There, we can perform the weaving procedurally by using the *XTend* model transformation language of oAW. For textual languages that do not provide means for proper SoC we also employ active artifacts, namely templates written in oAW's text generation language *XPand*.

SoC for Active Artifacts
The active logic of PLiCs also has to be integrated into the overall product generation process. As separate model transformation modules can often be serialized, there rarely is a need for advanced SoC mechanisms such as aspect orientation. It is sufficient to control the proper execution of the model transformation sequence from the top artifact level. This does however not apply to code generation. Variation points must be explicitly foreseen in generator templates to be able to customize certain parts of a generated file. To be able to customize code generator templates without invasive refactoring, we use the aspect-oriented extensions [26,36] of the XPand language of oAW.

Top Level Artifact
In oAW the overall product generation process (i. e., the sequence of model transformation and text generation) is defined in an XML dialect called *workflow language*. We currently do not employ techniques for advanced SoC at this level and assume that these issues are solved on lower artifact levels. We use the workflow language therefore at the top artifact level to integrate the various customizations of passive and active artifacts.

The customizations relate to the PLiC Reference Architecture (Figure 1) as follows: basic technologies such as transforming models, generating files, weaving aspect code, etc. reside in layer 0. The PLiC Framework on layer 1 provides a common interface to these, so they can be used as building blocks to compose complex product derivation. In our case, the building blocks are available as "functions" in the oAW workflow language, so that a software architect can concisely formulate the interfacing of product line components and the base product line via writing an oAW workflow.

We will now illustrate how flexible SoC-maintaining customizations can be implemented by describing concrete examples for metamodel (a special case of model weav-

ing), transformator, and generator customization. All examples have been implemented within the Safety PLiC case study using the above-mentioned techniques.

3.3.2. Metamodel Customization Example

SmartHome provides two different metamodels in the problem space: one feature model [21] for *configurative variability* (burglar alarm feature, temperature manager feature) and one EMF-based metamodel for *constructive variability* (for modeling rooms, doors, windows, sensor and actuator devices, controllers, etc.). Models (instances of the metamodels) describe a concrete house and are the input for product generation. To express additional safety concepts within the problem space models, the Safety PLiC extends the corresponding *meta*models. It provides the product line with additional problem space variability: for error injection (for testing purposes) and triple modular redundancy (TMR).

We created an additional safety feature model for the PLiC with only one optional feature: `errorinjection` (see Figure 2, left). Weaving with the base feature model is a function provided by the PLiC Framework and currently produces just a simple union, which is logically equivalent to two separate feature models. To formulate dependencies between features of the base product line and the PLiCs we use the constraint mechanisms of pure::variants [6], a commercial tool we use for feature modeling.[3]

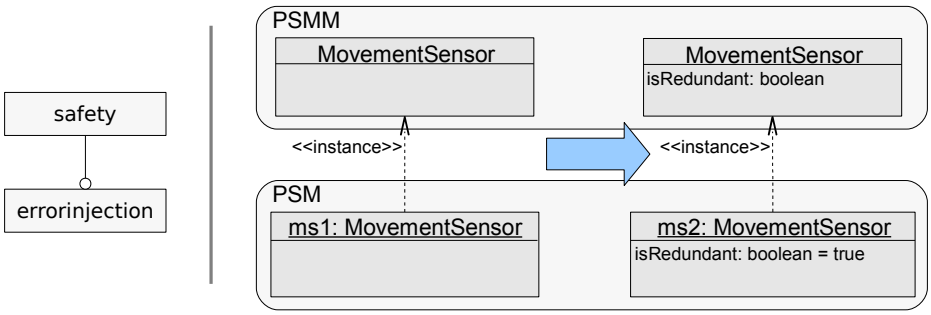

Figure 2. The Problem Space Metamodels of the Safety PLiC

Furthermore, we extended the EMF-based constructive variability metamodel by attaching the boolean attribute `isRedundant` to certain devices types. When set to true for a device, a procedural model transformation (Section 3.3.3) will generate the code for three device instances and a voter instead of a single device. The weaving in this case is performed by a procedural model transformation. The right-hand side of Figure 2 shows the adding of the `isRedundant` attribute to the metamodel element `MovementSensor`.

3.3.3. Transformator Customization Example

SmartHome maps the devices modeled in problem space to a platform-independent application component model in solution space. There, the PLiC Coordinator executes an

[3]Error injection is implemented as AspectJ aspect, which is a noninvasive code level customization. The aspect is woven only in case the `errorinjection` feature is selected.

additional model transformation provided by the Safety PLiC. If the `isRedundant` attribute is selected, the model transformation then triplicates the corresponding application component model elements, generates an appropriate voter component model element and wires them appropriately (Figure 3).

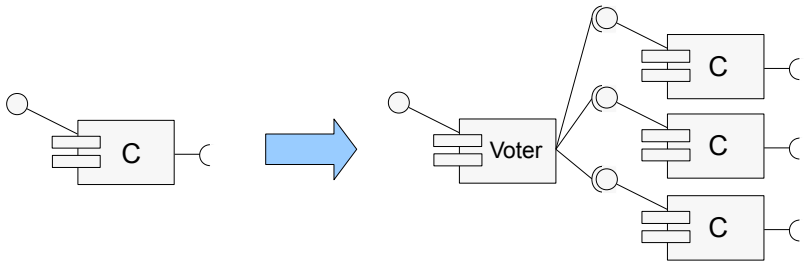

Figure 3. Transformation to a TMR System (Illustrative Example)

The model element of a voter component is generated to map exactly the interface of the encapsulated application component. This approach enables us to add TMR capabilities to virtually every application component. Note, however, that the decision if the semantics of an application component actually allows TMR capabilities is shifted to the application engineer.

3.3.4. Generator Customization

Since the model element of a voter component is generated, its implementation code must be generated as well. We do this with additional code generator templates the Safety PLiC provides. For every method of every voter component in the model, appropriate code is generated to invoke the encapsulated application components, collect their responses, perform majority voting and return the result or trigger error processing.

4. A Methodology for Integrating PLiCs With the Base Product Line

We already have shown how to map a product line to our reference architecture and how arbitrary types of customizations can be implemented while maintaining SoC. In this section, we will provide a methodology for integrating PLiCs into the base product line that also considers the derivation of a product from the extended product line. The PLiC Coordinator serves for automating this process.

4.1. Example Problem

The SmartHome product line is itself a product on the software market. As safety is optional, it must be acquirable with and without the Safety PLiC integrated. A methodology and a tool supporting the customization process are needed. It has to weave the concerns of the base product line and the PLiC to build up a composed product line that facilitates generating customized products.

4.2. The PLiC Coordinator

A PLiC Coordinator needs to be implemented for each base product line. It performs exactly three tasks: It (PCTask 1) *specifies the customization points* where PLiCs can extend the base product line, it provides functionality to (PCTask 2) *set up a new CPL*, and, when it is invoked by the CPL during product generation, it (PCTask 3) *mediates between base product line and PLiCs*, so that PLiCs can only extend the base product line as specified beforehand.

4.3. Methodology for integrating PLiCs

Before we describe how we implemented the PLiC Coordinator for the concrete Smart-Home case study, we give an overview of how to apply the general methodology. Therefore, we assume that a PLiC Coordinator and PLiCs already have been implemented. The following four steps need to be performed for generating a customized product (cf. Figure 4).

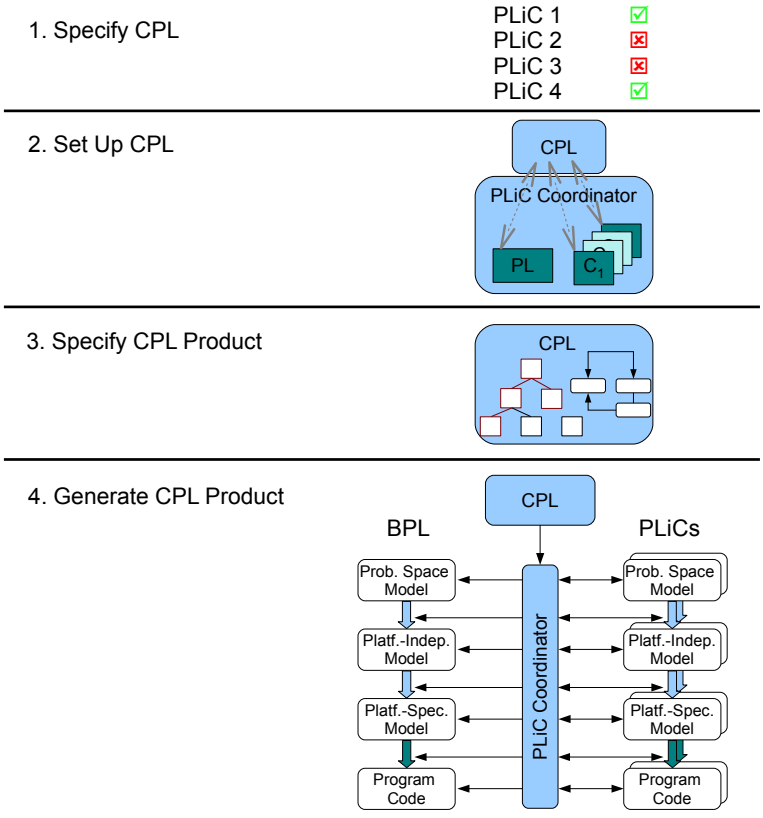

Figure 4. The PLiC Coordinator From a User Perspective

1. Specify CPL
   Given that the PLiC Coordinator, the base product line, and the PLiCs, are al-

ready implemented, a CPL needs to be specified. Therefore, a human configurator selects the PLiCs that shall customize the base product line.

2. Set Up CPL
   The PLiC Coordinator tool then sets up a CPL (PCTask 2) according to the CPL specification. Thereby, the CPL mainly consists of customized problem space metamodels, which have to be instantiated for product specification, and generation scripts that refer to the PLiC Coordinator.
3. Specify CPL Product
   An application engineer creates the problem space models based on the customized metamodels the CPL provides.
4. Generate CPL Product
   The CPL passes the problem space models to the PLiC Coordinator, which runs the product generation and integrates base product line and PLiC artifacts at all appropriate locations (PCTask 3, based on the specification of PCTask 1). Figure 4, phase 4, shows these locations for the SmartHome example case.

The PLiC Coordinator implements the customization phases two and four, whereas phases one and three are manual configuration steps. The details of the activities will differ depending on the languages employed (e.g., for modeling, model transformations, code generation, and programming) and on the desired granularity of customization capabilities. The subsequent section will explain this concept with SmartHome and the Safety PLiC example.

4.4. Example Solution

This section describes the implementation of the PLiC Coordinator of SmartHome. First, the tooling is described: the oAW workflow language and how the PLiC Framework extends it (corresponding to layer 0 and 1 in Figure 1). Then, we will illustrate how the PLiC Coordinator is implemented in the extended oAW workflow language and how our methodology has been applied to SmartHome and the Safety PLiC.

4.4.1. The oAW Workflow Language and the PLiC Framework

The oAW framework facilitates the specification of model-driven processes with an XML-based domain-specific language (*oAW workflow language*). In its basic version, it already supports loading of models and metamodels, scheduling model transformations and code generation, and integrating arbitrary other functionality with a lightweight plug-in mechanism called *workflow components*. The workflow language is the preferred way for oAW developers to specify the overall structure of the model-driven application generation process. Workflow scripts therefore reside on the top artifact level according to our classification in Section 3.2. Roughly, they may be characterized as Apache-Ant–like build scripts for model-driven application generation.

As a consequence, the PLiC Coordinator for our oAW-based product line is implemented in the workflow language. To ease defining PLiC Coordinators for other oAW-based product lines, the PLiC Framework we developed offers various additional workflow components. They constitute the building blocks to implement the three PLiC Coordinator tasks PCTask 1 to 3: specifying the customization points for PLiCs, creating CPLs, and mediating between PLiCs and base product line during product generation.

Actually, the first and the third task can be combined. For this purpose, one has to accept that the *script performing* the customization is at the same time the *specification* of the locations where the based product line may be customized.

4.4.2. Implementation of the PLiC Coordinator for SmartHome

We implemented all necessary tasks for SmartHome's PLiC Coordinator. Thereby, we decided to conflate PCTask 1 and 3, which basically means that the script performing the customization is at the same time the specification of allowed customization points.

For concise formulation, we developed an extension to the oAW language that supports iteration. For example, in the following Listing 1, the SmartHome PLiC Coordinator specifies that each PLiC needs to provide a file called `model.xmi` at a certain location in the file system. During runtime, it loads the model and adds it to a so-called list slot with name `modelLib` for further processing:

Assuming

```
<plicIterate order="luxuryplic,safetyplic">
 <plicEnv>
  <plicInjector
   parameter="uri"
   pattern="platform:/%{plicName}/src/model.xmi"/>
  <plicInjector
   parameter="modelSlot"
   pattern="%{plicName}"/>
  <readModel/>
 </plicEnv>
 <plicEnv>
  <plicInjector
   parameter="modelSlot"
   pattern="%{plicName}"/>
  <addToListSlot
   listSlot="modelLib"/>
 </plicEnv>
</plicIterate>
```

Listing 1: Iteration Example for oAW Workflow Language

The `plicIterate` element denotes a loop statement, which iterates over all PLiCs of the current CPL. It has an optional `order` attribute specifying the iteration order. The `plicIterate` element contains an arbitrary number of `plicEnv` environments. Each `plicEnv` environment denotes a parameterized call to a basic workflow component. Therefore, it contains a number of `plicInjector` elements, which characterize the parameterization, and one workflow component (in the example, `readModel` and `addToListSlot`, respectively), which is called with variable parameters. In the example, the first `plicEnv` environment will inject the values for the parameters `uri` and `modelSlot` into the `readModel` workflow component component. In each iteration these two parameters of the `readModel` component will be injected differently according to what is specified in the `pattern` attribute of an injector. The `%{plicName}` token within a pattern will be substituted with the actual name of the PLiC for each iteration. The second environment injects the name of the current PLiC

into the parameter `modelSlot` of the `addToListSlot` component accordingly. Patterns may also contain calls to static Java methods for more flexible pattern resolution, e.g., `%{mypack.MyClass.modelPath4PLiC(%{plicName})}` is also a valid token in a pattern.

All customizations PLiCs add to the base product line are implemented similarly in the oAW workflow language, for example, for adding further model transformations, source code, or metamodel weavers for CPL creation. As a general rule, PLiCs remain passive and provide their customizations as files in the file system following the naming convention implied by the PLiC Coordinator. This makes sure that PLiCs cannot act uncontrolled on the product line, but only as specified by the PLiC Coordinator.

4.4.3. Integrating SmartHome and the Safety PLiC

In the following, we apply our methodology thereby illustrating the steps required to integrate the Safety PLiC into the SmartHome product line (also cf. Figure 4).

1. Specify CPL
   A human configurator selects the Safety PLiC for integration into the CPL. Furthermore, he sets up the *integration configuration* of the Safety PLiC, which determines the way *how* the PLiC establishes the CPL.
   In our case study, for example, we want to control explicitly which of the SmartHome devices may later be transformed to a TMR system. The human configurator thus can explicitly select in the integration configuration those metamodel elements that shall have the `isRedundant` attribute, what allows for semantically correct problem space metamodel instantiation during CPL Product Specification. We support two kinds of integration configuration, via feature model and via textual configuration file.
2. Create CPL
   When its appropriate oAW workflow is executed the PLiC Coordinator creates the CPL. As specified in this workflow, the Safety PLiC may change the problem space metamodels of the base product line. It alters them in two different ways: the feature model is extended with an additional feature for `errorinjection` and the constructive variability metamodel is extended with an additional `isRedundant` attribute for all configured metamodel elements (e.g., the `MovementSensor`, see Figure 2).
3. Specify CPL Product
   The application engineer (i.e., a building architect) now manually configures the feature model and constructs a model conforming to the extended constructive variability metamodel according to his needs. For our case study, we assume that he actually activates the `errorinjection` feature and instantiates a building with some devices including a `MovementSensor` with its `isRedundant` attribute set to true.
4. Generate CPL Product
   For product generation, the CPL redirects the actual integration to the workflow of the PLiC Coordinator. The PLiC Coordinator takes care of executing the transformators and generators of SmartHome and the Safety PLiC at the appropriate stages of the product generation process. Furthermore, it arranges the adding of further model elements and of manually written application code provided by the PLiC at appropriate locations in the file system.

In our use case, we execute the additional transformator and generators of the Safety PLiC as stated in Section 3.3. We also include the model elements and the manually implemented code. The final result is a GUI-based application with functionality for injecting errors and testing the functionality of the voter devices.

5. Further Study Results

This section will give more details on the case study performed. As a result, we can say that customizing the product line on various levels with our approach has been possible with very reasonable additional effort (less than 1000 lines of new and re-factored code, implemented in about 3 man-days). This gives evidence that our approach can be applied in various other contexts where the same or fewer customization levels need to be considered. As the PLiC Framework layer needs to be implemented only once per base technology (e.g., oAW), the additional effort per product line is threefold: refactoring the base product line, implementing a PLiC Coordinator and CPLs, and implementing customizations in separate PLiCs instead of intermingling their functionality with the base product line.

5.1. Refactoring the Base Product Line SmartHome

We implemented two variants of the PLiC Approach for SmartHome. For the first alternative we ported the workflow files of SmartHome to the PLiC Coordinator and introduced the ability to iterate over PLiC artifacts at appropriate locations. The refactoring affected approx. 700 lines of workflow code (size of an average product is around 20K lines in Java).

Second we implemented a minimal-invasive version, in which the PLiC Coordinator uses the aspect-oriented capabilities of the oAW framework and our extensions provided by the PLiC Framework to weave the corresponding PLiC artifacts into the base SmartHome workflow at the appropriate locations. Doing so, we could restrict the changes to a few lines of code and the SmartHome product line still had no direct dependency on the PLiC Framework, making its use and thus customization fully optional.

Comparing the two we prefer the first refactoring alternative. Although it initially resulted in more effort, we expect it to cope better with product line evolution. Aspect-orientation, as used in the second alternative, often suffers from the fragile pointcut problem [32], which occurs when a change in the base program has an unintended effect on the set of points in the control flow (join points) an aspect gives advice to.

5.2. Implementing the PLiC Coordinator and CPLs

The oAW workflow language and our extension to it provide a good level of abstraction. Therefore, the implementation of a PLiC Coordinator can be kept very concise. The PLiC Coordinator only consists of workflow files of about 140 lines of XML in total in the aspect-oriented version and 850 lines in the version where we ported all workflow files from SmartHome to the PLiC Coordinator. The CPL for SmartHome plus the Safety PLiC comprises less then 40 lines written in the workflow language.

5.3. PLiC Implementation

We developed two PLiCs for the SmartHome product line, one for luxury and one for industrial settings.

The *Industrial PLiC* provides a superset of the functionality of the Safety PLiC, which we used to present our approach throughout this paper. It additionally introduces a monitoring interface to all application components on model level and generates the appropriate application code into the base classes of the components. Furthermore, an additional monitoring station component is introduced into the system polling the status of the devices and reacting accordingly (warning, alarm, emergency shut-down, etc.).

The *Luxury PLiC* introduces convenience devices and their implementation. The `HumanPresenceSensor`, for instance, detects the presence of residents. According to their profile, the convenience functions (temperature, light, windows, etc.) can vary. Touchscreen devices, which can be placed into the rooms of the house, allow the management of the profiles. An HTTP interface facilitates control of house settings with general-purpose computers or handheld devices.

Both PLiCs have been implemented maintaining strict SoC between their functionality and the base product line. For this purpose, we used the customization strategy and the publicly available technologies and tools explained in Section 3.3. By providing a *template PLiC*, we were able to minimize the additional effort when creating a new PLiC. The template PLiC already provides "empty" metamodel, model, transformation, and generator files as well as an empty source code folder at the locations the PLiC Coordinator expects them. When developing a new PLiC based on the template, one can directly add the new model elements or execution logic to the appropriate files and the PLiC Coordinator will immediately consider them without further effort.

6. Discussion

In this section, we discuss the applicability of the approach in other contexts, describe the types of variability our approach is suited for, and address the scalability of the approach.

6.1. Applicability in Other Contexts

The PLiC Approach is not limited to certain implementation languages or capabilities. In Section 3.2, we already have indicated how separation of concerns can be handled depending on the capabilities of the employed implementation languages and the granularity of customizations. Comparing the three classes of customizations, we favor the use of noninvasive or invasive techniques over customization by shifting the artifact level from passive to active artifacts. Whilst we do not see scalability issues when using model transformators, text generation templates makes the generated code more complex to deal with and should therefore not be used extensively, merely for customization purposes.

As a consequence we consider the PLiC Approach as not suited when the following conditions come together: (1) fine-grained product line customizations must be scattered over a (2) large number of files having (3) languages with limited means for SoC and (4) design patterns for loose-coupling may not be applied. A purely preprocessor-based repository approach as described in Section 1 is more appropriate in such a case, although

it bears the indicated shortcomings. Note, however, that the PLiC Approach does not prescribe how to implement variability *within* the context *local* to a PLiC or the base product line. In particular, concerns may be intermingled and even be implemented by preprocessors statements there.

Our approach is not limited to model-driven product lines; in our case study we also dealt with manually implemented code. Having a non-model-driven process would even ease product generation to some extent, as there is no staged modeling process with several engagement points for PLiCs on different levels. For primarily manually implemented product lines, the user would usually exchange the oAW workflow language, which is very useful to specify model-driven processes, with other build languages, for example make files or Ant scripts. Arbitrary build languages are possible, but they have to support parameterizable functional abstraction and an iteration concept. Although these capabilities are not really the strength of make files and of Ant scripts, they can be integrated when using shell scripting within make and Java Ant Tasks within Ant.

As mentioned, we build on an extension of the oAW workflow engine for our approach. The extension became necessary due to its restricted functional and iteration capabilities. In Section 8, we will indicate how we intend to further improve this situation.

6.2. Supported Variability Types

Variability generally can be implemented in a positive or negative manner, that is either a variant is built in and cut out if not needed (negative), or a minimal core is extended with an option (positive). We do not recommend the use of PLiCs to implement negative variability with respect to the base product line, however. PLiCs mainly induce additive functionality on model level with additional transformator and generator logic, additive functionality on code level by adding classes and aspects. Although it is possible to remove functionality from the base product line by adding a PLiC with an aspect on code level or model level, we agree with GACEK ET AL. [16] that doing this with aspect orientation can be an inefficient and difficult task and might require unorthodox refactorings within the base product line.

6.3. Scalability

We do not have scalability studies beyond the integration of the two PLiCs mentioned. The concept is intended to include or exclude specific sub-domains according to customer needs, so we expect rather few PLiCs. In its integration feature model (see Section 4.4.3), a PLiC can define dependencies regarding other PLiCs. Although we can technically handle a larger amount of PLiCs this way, we recommend the use of lightweight features over heavyweight PLiCs in cases where product line customization is not needed.

Testing efforts increase when using product line customization. The features the PLiCs introduce have to be tested in various combinations, of course. Additionally, all valid combinations of base product line and PLiCs have to be tested. However, this is an inherent property of product line variability and not a shortcoming of the PLiC Approach.

7. Related Work

The PLiC Approach defines a reference architecture and a method for customizing product lines on multiple levels. This is a new viewpoint; to our knowledge there exist no publications regarding this topic. There are, however, various points of intersection with related work, in particular when we account for the abilities we consider necessary: We developed it to support full product derivation automation, to offer flexible and comprehensive means of customizations, and to be tailorable to a wide range of product line settings. There are no other approaches that fulfill all these requirements we regard as necessary.

Our work was initially inspired by so called *hierarchical product families*, which consist of several product families built upon a common core product family platform (BOSCH, [9]). The approach usually implies extensive manual effort during platform customization and product derivation. The PLiC Approach combines the hierarchical idea with automated product lines, which are specified only in problem space, model-driven product derivation, and highly flexible customization techniques.

In [13], CZARNECKI ET AL. present their concept for *multi-level customization in application engineering*. Our approach resembles it as we also identify the need for customizing domain-specific model-driven product line platforms. Whereas the authors concentrate on feature-based customization of models, we also cover transformator, generator, and code level customizations. Furthermore, we do not tackle the customization of products in application engineering, but the customization of product *lines* in *domain engineering*. In particular we consider extending problem space metamodels and therefore the product line domain.

BOSCH furthermore discusses *composition-oriented approaches* in [9]. They are based on components that are freely composable within specified architecture composition rules. When the components are structured hierarchically and still offer means to defer variability binding to later stages, like in the *product populations* approach of OMMERING [34], they could be used to implement similar customization objectives. However, the approach is usually tied to the Koala [35] component model, which determines the implementation style and language and can therefore not be applied in arbitrary product line settings. Furthermore it neither addresses model-driven development nor separation of concerns.

AHEAD of BATORY ET AL. [4,5] and its model-driven extension [33] derive products by means of step-wise refinement. The authors do not tackle the problem of customizing model-driven product lines in domain engineering; nevertheless their work results relevant to our research. The concept of feature-oriented model-driven development (FOMDD) includes *feature refinements*, which refine *artifacts* (e.g., program code or models) with the feature-oriented superimposition mechanism, and *model transformations*, which transform artifacts (model-to-model, model-to-text, text-to-text). Thus, a variable set of artifacts and feature refinements, together with the static set of model transformations, constitutes a product line. Providing different subsets of artifacts and feature refinements to different customers then would conform to what we call product line customization.

FOMDD postulates a product line to exhibit strong properties: implementing each feature in separate files and having a commuting relationship between features refinements and model transformations [33]. Both properties are not respected within the PLiC

Approach and we expect them to hinder scalability and expressivity. Having a commuting relationship between feature refinements and model transformations means that every refinement of models or code in solution space *must* also already be possible in problem space and the problem space refinement *must* be transformable to the solution space feature refinement. This restriction either (1) hinders concise, domain-specific problem space models or (2) hinders refinement flexibility in solution space models. In any case, it (3) limits the expressivity of refinements, as they themselves must be transformable. On the other hand, implementing each feature strictly separately in a model-driven environment would lead to a fast increase of the number of files and to complex dependencies for non trivial features. The PLiC Approach, in contrast, bundles several features according to their sub-domains into distinct PLiCs. This means, a PLiC can provide more coarse-grained means for composing product lines leading to less and less intricate dependencies during composition, as within the context of a PLiC, we are not restricted to implement each feature in strictly separated artifact.

When implementing product lines strictly feature oriented and using certain restricted programming languages, recent research has shown how type safety can be guaranteed for all valid feature compositions [2]. As our approach intentionally does not make this restrictions, we cannot leverage their results. However, one promising direction might be to restrain ourselves to component-based composition and apply an approach similar to [20], where the authors extend various existing component models (e.g., EJB, and CORBA) with type checking.

There are aspect-oriented modeling approaches which primarily concentrate on separating concerns in UML-based solution space models and therefore have a different focus. They do not provide a methodology for flexible customization, but rather an approach for composing model fragments. Theme/UML [3,11] uses UML aspect models to describe different concerns of an application and transforms them to an aspect-oriented programming language such as AspectJ. The aspect-oriented model-driven framework described in [29], in contrast, performs the weaving of the aspect models already on model level, so that code for arbitrary programming languages may be generated. Both aspect model composition approaches fit well into the PLiC Approach, where they would be applied in a similar manner as the model weaver XWeave [18] we have used in our SmartHome case study.

One refinement approach for product lines specifically tackling model-driven development is that of YIE ET AL. [39]. There exists a concept comparable to PLiCs called model refinement line (MRL). The authors propose the use of separate, aspect-oriented models for MRLs along several model-driven stages. A high-level security aspect model of the MRL, together with its references to the base problem space model, is transformed separately from the base product line. The actual integration of the MRL model and the product line model is deferred to the platform-specific level right before code generation. On the one hand, the approach is limited to customization in problem space, as the aspect-oriented security model only can reference concepts existing in the base problem space model. On the other hand, additional model transformations must be implemented for the security aspect model to transform it to a platform-specific solution space representation until it can be woven into the base product line.

A further issue to consider is the collaboration of the various stakeholders participating in large-scale product-line configuration. In [24], the authors propose to use a workflow-like plan that safely guides stakeholders during the configuration process and

propose a set of reasoning algorithms that can be used to provide automated support for product configuration. Currently, our approach only addresses the implementation side of product line composition not reasoning and process support for configuration. The integration of these mechanisms and algorithms was, at least for this paper, out of scope.

Summing up, all related approaches supporting automated product derivation have rather limited means of customization. Each of them relies on one specific customization principle, whether composition based on model weaving [13], feature-oriented [33], aspect-oriented [29,11,39], or component-oriented [34] composition. They neglect the power of plurality of customization methods that we pointed out in Section 3 and that we regard as on of the key features of our approach.

8. Conclusion

In this paper we presented the PLiC Approach, which facilitates customizing whole product lines by slicing them into recombinable parts called product line components (PLiCs). Its generic reference architecture and the presented methodology are adaptable to a wide range of product line settings, consider flexible and comprehensive means of customizations, and provide for full product derivation automation. Our case study involving a complex model-driven product line has been implemented with reasonable effort, while maintaining strict separation of concerns between base product line and PLiCs. It gives evidence for the applicability of our approach to product lines having equally or less complex product derivation infrastructures.

For comprehensive customization, a modular and well-considered product derivation architecture is crucial. The next step we plan to take is therefore designing an aspect-oriented extension for the oAW workflow language [15] to provide also the topmost level of a model-driven product derivation infrastructure with capabilities for SoC. Finally, we hope that our point of view will help us to develop better patterns for product derivation infrastructures in general. Modularization and separation of concerns have multiple benefits on this level, not only for the purpose of customization.

References

[1] AMPLE project homepage. Aspect-Oriented Model-Driven Product Line Engineering. http://ample.holos.pt/.

[2] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering – An International Journal.*

[3] Elisa Baniassad and Siobhán Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society Press.

[4] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.

[5] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society Press.

[6] Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, visited 2009-03-26.

[7] Jan Bosch. *Design and Use of Software Architectures, Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[8] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the 2nd Software Product Line Conference (SPLC '02)*, pages 257–271, Heidelberg, Germany, 2002. Springer-Verlag.

[9] Jan Bosch. Expanding the scope of software product families: Problems and alternative approaches. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[10] Jan Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, 2009. ISBN 978-0-9786956-2-0.

[11] Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, Boston, 2005.

[12] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.

[13] Krzysztof Czarnecki, Michal Antkiewicz, and Chang Hwan Peter Kim. Multi-level customization in application engineering. *Communications of the ACM, Special Issue on Software-Product Line Engineering*, pages 61–66, December 2006.

[14] Christoph Elsner, Ludger Fiege, Iris Groher, Michael Jäger, Christa Schwanninger, and Markus Völter. Ample deliverable 5.3: Implementation of first case study: Smart home. `http://ample.holos.pt/gest_cnt_upload/editor/File/public/Deliverable%20D5.3.doc`.

[15] Christoph Elsner, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Towards separation of concerns in model transformation workflows. In Steffen Thiel and Klaus Pohl, editors, *Proceedings of the 12th Software Product Line Conference (SPLC '08), Second Volume*. Lero International Science Centre, 2008.

[16] Cristina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR '01)*, pages 109–117. ACM Press, 2001.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] Iris Groher and Markus Völter. XWeave: Models and aspects in concert. *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*, pages 35–40, 2007.

[19] Herman Hartmann, Tim Trew, and Aart Matsinger. Supplier independent feature modelling. In *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, 2009. ISBN 978-0-9786956-2-0.

[20] Georg Jung and John Hatcliff. A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures. *Science of Computer Programming*, 75(7):615–637, July 2010.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.

[23] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[24] Marcilio Mendonca and Donald Cowan. Decision-making coordination and efficient reasoning techniques for feature-based configuration. *Science of Computer Programming*, 75(5):311–332, May 2010.

[25] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[26] OpenArchitectureWare homepage. `http://www.openarchitectureware.org/`.

[27] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[28] ProOSEK homepage. `http://www.proosek.de/`.

[29] Devon Simmonds, Raghu Reddy, Robert France, Sudipto Ghosh, and Arnor Solberg. An aspect oriented model driven framework. In *Proceedings of the 9th IEEE International EDOC Conference*, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[31] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[32] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[33] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 44–53, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Rob van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 255–265, New York, NY, USA, 2002. ACM Press.

[35] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[36] Markus Völter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 233–242, 2007.

[37] The Wind River VxWorks homepage. `http://cdn.windriver.com/products/vxworks/`.

[38] The Windows embedded CE developer center. `http://msdn.microsoft.com/en-us/embedded/aa731407.aspx`.

[39] Andres Yie, Rubby Casallas, Dirk Deridder, and Ragnhild Van Der Straeten. Multi-step concern refinement. In *Proceedings of the Workshop on Early Aspects (AOSD-EA '08)*, 2008.