Multi-Variability Modeling and Realization for Software Derivation in Industrial Automation Management

Miao Fang
University of Kaiserslautern
Kaiserslautern, Germany
miaofang@rhrk.uni-kl.de

Georg Leyh
Siemens AG
Erlangen, Germany
georg.leyh@siemens.com

Joerg Doerr
Fraunhofer Institute IESE
Kaiserslautern, Germany
joerg.doerr@iese.fraunhofer.de

Christoph Elsner
Siemens AG
Erlangen, Germany
christoph.elsner@siemens.com

ABSTRACT

The systems of industrial automation management (IAM) are in the domain of information systems. IAM systems have software components that support manufacturing processes. The operational parts of IAM coordinate highly plug-compatible hardware devices. These functions of the IAM systems lead to process and topology variability, which result in development and reuse challenges for software engineers in practice. This paper presents an approach aiming at improving the development and derivation of one IAM software family within Siemens. The approach integrates feature modeling with domain-specific modeling languages (DSMLs) for variability representation. Moreover, by combining code generation techniques, the configuration of variability models can be used to automate the software derivation. We report a case study of applying the approach in practice. The outcome shows the enhancement of variability representation by introducing DSMLs and the improvement on automating software derivation. Finally, we report the lessons learned during the execution of this case study.

Keywords

Model-Based Engineering, Domain-Specific Modeling, Software Product Line, Variability Modeling, Software Derivation, Code Generation

1. INTRODUCTION

Large-scale projects gain more attention in the realm of software and systems engineering. Highly integrated software and hardware devices enable more efficient IT solutions, and at the same time bring high complexities to development. In the context of this paper, we focus on the domain of industrial automation management (IAM). These systems are information systems managing manufacturing processes and coordinating distributed hardware devices in factories, for example for automobile or food production. The development and reuse of software in this domain is very challenging, as the implementation is influenced by many different factors, from customer requirements to constraints of hardware distribution.

Software product line engineering (SPLE) is a set of techniques for managing and maintaining core assets and systematically reusing them during the development of new software products. A wide variety of companies and organizations have adopted SPLE and reported substantial benefits, such as reduced development costs and shortened time-to-market [6]. With this paper, we report how SPLE can be used in an IAM setting with the necessary adaptation to support variability modeling and product derivation. The ultimate goal is to help the development team to save time and effort when developing new software systems.

SPLE can be characterized by two processes, family engineering (also known as domain engineering) and application engineering [15]. During family engineering, commonalities and variability of a product family are identified. Reusable artifacts are developed and managed as core assets, such as requirement scenarios, reusable architectures, and reusable components. During application engineering, the reusable assets are selected and configured by using variability models to fulfill customer-specific requirements. Variability is one of the core concepts in SPLE, as it describes how products can vary among one another. Many variability modeling techniques have been proposed and used in academia and practice, for example feature modeling, decision modeling, and orthogonal variability modeling [8].

However, IAM systems have process and topology variability, which cause difficulties in variability modeling and representation. General-purpose variability techniques, such as feature modeling, have their limitations to express these two domain variability types [12]. In IAM systems, the process variability stems from the variable manufacturing processes required by customers, and the topology variability comes from the flexible configuration of hardware devices in the manufacturing factories. The current IAM development in practice is tedious and time consuming. Using domain-specific models during IAM development and derivation is beneficial for the stakeholders' satisfaction and efficiency, as we pointed out in one of our previous studies [13].

MODELS '16, October 02 - 07, 2016, Saint-Malo, France

ISBN 978-1-4503-4321-3/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976767.2976804

In this paper, we present a model-based approach, which integrates feature modeling with domain-specific modeling languages (DSMLs) for expressing processes and topology. The three types of variability models are interrelated and realized in the technical solution space. This enables us to provide semi-automated support to derive software architecture and to generate source code. We report on a case study of applying the approach to a chosen software family of warehouse management systems in Siemens. The results of the case study show that using DSMLs brings important benefits to achieve user-centered variability representation. Additionally, integrating DSMLs into derivation enables more fine-grained reuse.

The contributions of this paper are twofold. Firstly, we propose a multi-model-based engineering approach in a complex software domain. We combine the multiple models and implement them with code generation techniques, in order to save the effort of software development. Secondly, we report on the results of applying the approach in an industrial-grade case study, which was based on variability representation, feasibility and achievement of code generation. Besides, we also present our lessons learns during this study.

2. MOTIVATION

The motivation for this paper comes from the lack of satisfaction during development reported by software architects and developers of IAM systems. The products to be delivered to customers usually constitute an entire solution for a factory. Such a solution includes software and hardware systems, and covers the whole life cycle, from analysis, design, and development to installation and maintenance. IAM systems play both a role in implementing the customers' business and in managing underling software and hardware systems. In this case, customers are not the only sources where requirements come from. The hardware decisions and restrictions also have a strong impact on the software implementation. Software engineers are expected to understand and to cope with all these requirements and restrictions, as a result, they feel highly challenged during development.

To approach this problem, we chose a software product family developed in Siemens, warehouse management systems, as a case study to implement a model-based approach for IAM systems. Warehouse management systems are a typical sub-domain of industrial automation management. Manufacturers, importers, exporters, and logistics businesses use warehouses for the storage of goods or products. Warehouse management systems have process-like functions, such as goods-in, storage, picking, packing and shipping. Figure 1 shows the functional zones of performing these tasks in a small warehouse layout. The arrows in this layout illustrate the material flows, which are commonly used in the IAM domain to represent how manufacturing materials are transported, stored, supplied, and assembled in factories.

2.1 Multi-Variability Types for Modeling

Variability is the "ability of a software system or artifact to be efficiently extended, changed, customized or configured to be used in a particular context" [24]. Application engineering activities rely on the configuration of variability. Based on the configuration of variability models, the new software product can be derived efficiently to fulfill the customer requirements.

Many variability modeling techniques have been proposed

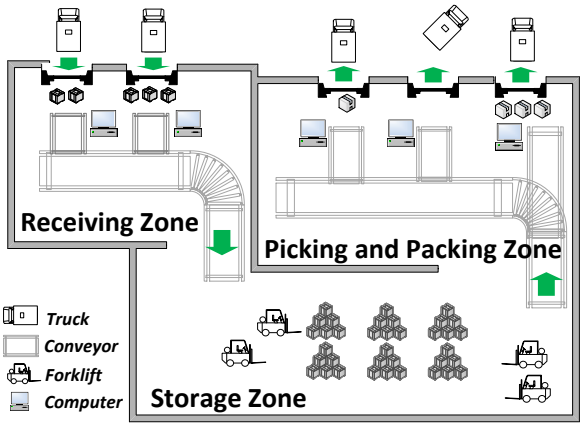

Figure 1: A Factory-Layout Example and Material Flows

and used in academia and practice [8]. Among all existing variability modeling techniques, feature and decision modeling have attracted most attention [10]. The feature or decision models are able to represent similar variants, such as mandatory, optional, alternative, multiplicity, and cardinality. In the context of this study, we address these variability as **feature-like variability**. The IAM systems have such feature-like variability, as it is common for many other software product lines as well. Besides, the IAM systems also have topology and process variability.

Topology variability. Figure 2 shows simplified layouts of the receiving zones in warehouse plants of two different customers. On the left side, *Customer 1* has an automatic goods-in forklift and a manual goods-in workstation, whereas on the right side, *Customer 2* has at least three manual goods-in workstations. As can be seen, various devices are installed at different locations of the plant floors. They are highly reusable and plug-compatible among customers. The variability is caused by the flexible combination of all these devices, since all the devices ideally can be wired directly or connected via communication networks.

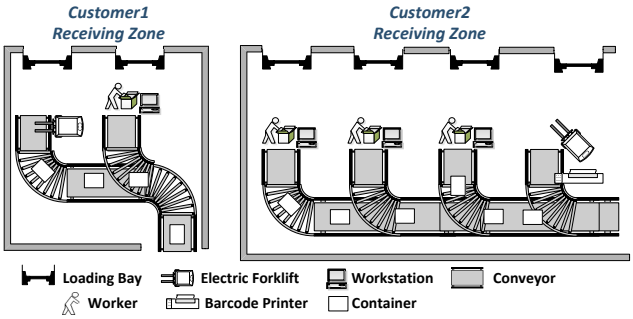

Figure 2: Topology Variability among Customers

Process variability. In the example above, the two customers have two different topology configurations, and they both require a manual goods-in process in their factories. Figure 3 shows an informal process description, shared among customers. The *Quality Control* is an optional activity in this process. In addition, the *Receive Goods' Notification* activity has to be bound to the particular topological configuration of each customer project. Only when the software system receives the notification of box arrivals at the corresponding locations, this process can be triggered.

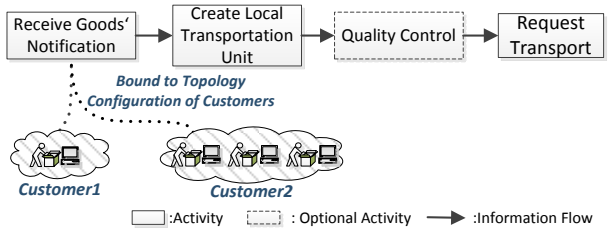

Figure 3: Process Descriptions and Relations to Topology

The topology and process variability are domain-specific variability types. They lead to difficulties, when we try to use general-purpose variability modeling techniques to express them [12]. To come up with a systematic SPLE approach for the target domain, it is important to model the topology and process variability and their inter-relationships.

2.2 Derivation Process and Stakeholders

Software product derivation includes the activities of selecting and configuring existing artifacts to create new products to satisfy customer-specific requirements. Feature modeling is often used to enable the software product derivation.

However, the product derivation in IAM has its special sequence. We identify three roles of derivation stakeholders, including requirement engineers, hardware-oriented engineers, and software developers. These three roles of stakeholders participate in a sequential derivation process for IAM development:

1. **Requirement engineers** focus mostly on process variability. Processes, as informal behavioral models, describe the behavior of systems that are closer to the material flows, which reflect directly what should happen in the real-world plants.
2. **Hardware-oriented engineers** have topology variability concerns. They are usually key engineers not only from hardware teams, but also from the requirement and development teams in practice settings. They join meetings together to communicate the topology configurations. The distributed hardware devices and the IT information systems need to be integrated at the end to perform the actual manufacturing tasks. For this reason, the hardware-oriented engineers serve as a "bridge" between the software and hardware teams.
3. **Software developers** have to understand the other two roles of stakeholders. They also have to understand the process and topology configurations of the desired applications, in order to bring them together and to implement the final IAM systems.

2.3 Challenge Analysis

Since software families of IAM systems have considerable commonalities, we expect that introducing SPLE techniques to enable reuse can bring significant benefits to the software development organizations. To develop a systematic approach, it is necessary to address two main challenges: (1) the need for user-centered variability modeling and (2) the need for automating the derivation process.

Firstly, an SPLE approach of IAM should support the representation of variability in the languages of the product derivation stakeholders [22]. Model-based techniques, in particular domain-specific modeling languages (DSMLs), specify software systems at the level of abstraction beyond code, using a notation closer to the problem space [28]. Meanwhile, using tailored "languages" to the domain can potentially bring the benefits of improving communication among the three identified stakeholders [25, 19]. With these expected benefits, domain-specific modeling languages are desired to be integrated in the approach to express the process and topology variability, to achieve user-centered variability modeling.

Secondly, when multiple kinds of variability models are involved for the configuration of the systems during application engineering time, it is important to integrate them systematically into the sequential derivation process. By doing so, the derivation process can take the advantages of the multi-variability models to achieve a higher level of automation and better efficiency.

3. APPROACH

Figure 4 presents the approach outline. It has three main elements: Multi-Variability Modeling, Multi-Variability Realization, and the integrated Derivation Process. We briefly describe these three main elements. The next three subsections address each of them in detail, by applying them within our warehouse management system case study:

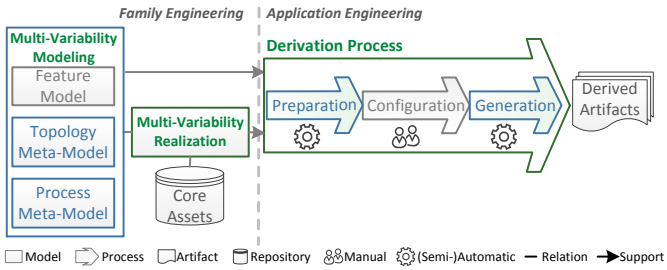

Figure 4: The Approach Outline

1. The *multi-variability modeling* involves three required variability types, their meta-models and the linkage among meta-elements.
2. The *multi-variability realization* addresses how variability is implemented technically, so that software artifacts can be selected and configured by using variability models. For this, the meta-elements in the variability models need to be associated with software artifacts, such as components and source code in the product line core assets.
3. The *semi-automated derivation process* can be further split into three phases in our approach: preparation, configuration, and generation as presented in Figure 4. The manual configuration phase is tool-supported, which allows stakeholders to focus on their own variability concerns and configuration

Figure 4 also presents that two of these solution parts are in family engineering, and the derivation process is in application engineering. But, the development of the derivation infrastructure supporting the derivation process is also done in family engineering. It is typical in SPLE approaches to invest more effort in family engineering, and to gain productivity and reuse in application engineering.

The initial creation, development and industrial utilization of the approach is aspired for warehouse management and automobile manufacturing. In the context of this study, we concentrate on enabling this approach for the chosen

warehouse case study in Siemens. The reason is that the developed DSMLs, language editors, and variability realization are domain-specific.

3.1 Multi-Variability Modeling

Figure 5 presents the architecture of the multi-variability modeling framework, according to the four-layered architecture standardized by Meta-Object Facility (MOF) [2]. The framework includes feature modeling for representing general software commonalities and variability, and two DSMLs for representing topology and processes. We introduce the developed meta-models within the M2 layer in the remainder of this section.

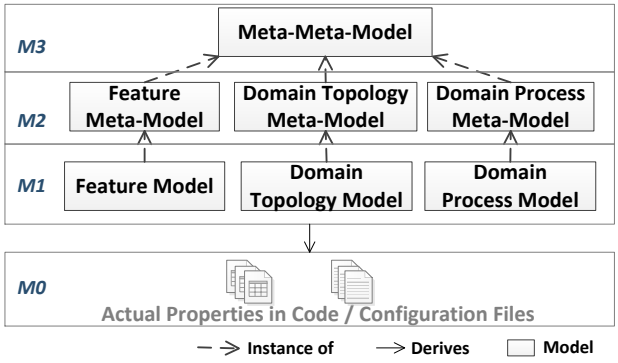

Figure 5: Multi-Variability Modeling

3.1.1 Feature Meta-Modeling

A feature model is a tree structure that consists of a set of feature nodes and a set of relations among the feature nodes. Since feature modeling is a de-facto variability modeling technique [9], many researchers propose available meta-models as "dialects" of feature modeling. Figure 6a presents an excerpt of the feature meta-model used in our approach. It consists of *Features* and *Feature Relationships* (the concrete sub-types of feature relationships, i.e., mandatory, optional, alternative, OR relationship, are left out for brevity). Figure 6b shows the typical notations for feature trees. The different relationship types are represented as open and filled circles and arcs.

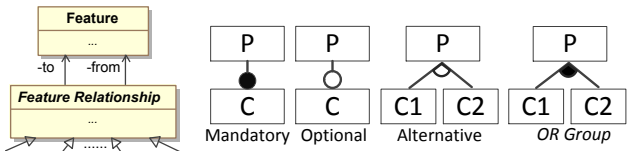

Figure 6: An Excerpt of the Feature Meta-Model, Relationships and Feature Notations

3.1.2 Topology Meta-Modeling

A topology model comprises a set of topological elements and their relationships within a given zone of a factory. The developed topology meta-model has two parts: a core meta-model and a supplementary meta-model.

Figure 7 shows the core meta-model, in which a topology model may have arbitrary numbers of topology elements. A *Topology Element* can be assigned to another element. The *Transporter Element* and *Stationary Element* are the two sub-types of *Topology Element*. Transporters refer to the transporting elements in factories such as conveyors to bring the working units from one location to another. Stationary elements are usually fixed devices, which actually perform concrete manufacturing tasks in factories. These manufacturing tasks can be fully automated, semi-automated, or manually performed by human workers.

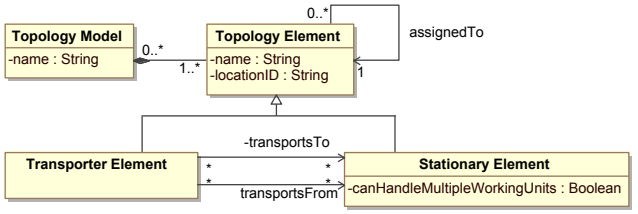

Figure 7: The Core Topology Meta-Model

Figure 8 presents an excerpt of the supplementary topology meta-model, which includes the sub-types *Transporter Element* and *Stationary Element*. These supplementary elements are sub-domain-specific for warehouse management systems. Other sub-domains (e.g., automotive or food production) may share some of these elements, and they may also require other elements.

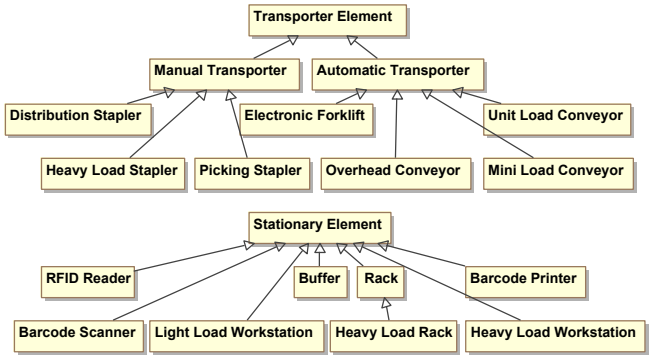

Figure 8: An Excerpt of the Supplementary Meta-Model

3.1.3 Process Meta-Modeling

A process model contains a set of action nodes and edges, which captures the behavioral flow of a system function. Figure 9 presents the developed process meta-model, in which we distinguish types of actions to describe processes. There are actions that are typical for information systems, such as *Computational Action* and *Human-System Interaction*. There are also several domain-specific actions. A *Manual Action* can be used to model manual steps performed by human workers. This is typical for the non-automated tasks in manufacturing factories. The *Automatic Action* is defined for representing the automated tasks, commonly coordinated or triggered by *Software Actions*, but actually performed by hardware devices. The *Periodic Action* has usually a loop time to re-execute.

3.1.4 Associating Meta-Models Hierarchically

Establishing multiple (domain-)variability models for representation purposes is not the ultimate goal of this approach. The associations among them are necessary to have,

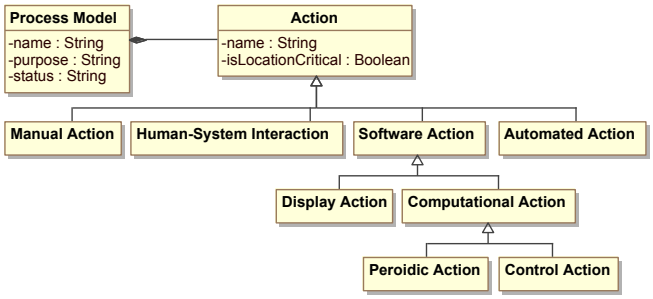

Figure 9: The Process Meta-Model

so that they can be systematically instantiated and integrated in the derivation process. Figure 10 shows how the feature, process, and topology (meta-)elements are associated in our approach.

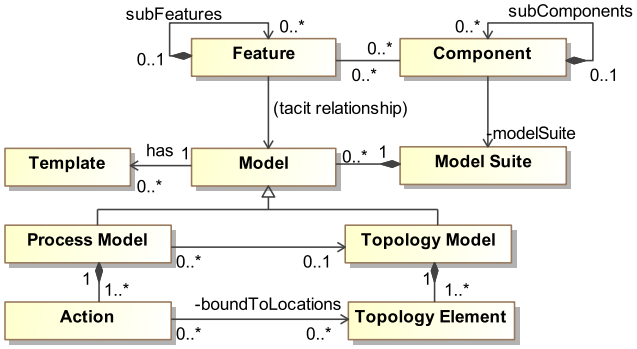

Figure 10: Hierarchical Multi-Variability Associations

Features may contain sub-features hierarchically. They are related to components and sub-components, which describe the software product line architecture. When a component has further process or topology variability, the corresponding variability models are added to a model suite. Such a model suite contains the information about the names, types of the variability models. Each involved model can have model templates, which are extracted during family engineering. During application engineering these models are instantiated with appropriate templates, to serve as base configurations to help derivation stakeholders to achieve better configuration productivity.

There can also be tacit relations and dependencies among *Features* and domain-specific variability *Models*. For example, a feature may imply a weak suggestion of another feature—or may require a more detailed description in a process model configuration. Existing works already propose solutions to solve these types of relations [30, 14]. In our target systems, we identify domain-specific relations between the process and topology models. In process models, there can be certain actions that are bound to certain topological locations (see the example in Figure 3). To describe this situation in models, we need the *boundToLocations* relation between the *Action* and the *Topology Element* as shown in Figure 10.

3.2 Multi-Variability Realization

Based on the established multi-variability meta-models and their associations, we further present the realization of the multi-variability of our approach in the technical implementation space. With the variability realization, reusable software artifacts can be finally selected and configured by instantiating and configuring variability models.

Variability realization can appear at two different levels of abstraction [29]:

- Architectural level: The typical reusable assets include the product line architecture, components, and frameworks. The variants are possibly optional components or architectural reorganizations [24].
- Code level: The commonalities are sets of classes in software product lines. The variable parts are "embedded" in code fragments [29, 24], that need to be composed to realize the variability.

At the architectural level, the variability realization of our targeted domain is similar to other software product lines. Feature-like variability at this level plays the most important role for component inclusion or exclusion, suggested by some existing approaches such as [29]. Figure 11 shows a small example about how feature-like variability is usually resolved during architecture derivation. *Pick by Light* is an optional component. When the corresponding feature is selected in the configuration, the derivation infrastructure includes the source of *Pick by Light* during architecture derivation, and wires up its instance to the appropriate instance of *Pick Logic* component. In this example, the parameters of components, such as the *mode* of the *Pick Logic* can be configured via selecting and configuring the value of the corresponding feature as well.

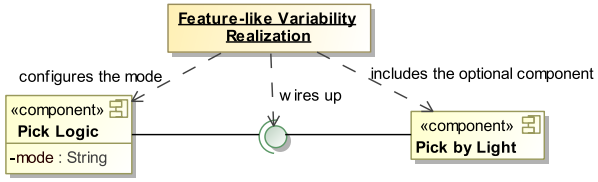

Figure 11: Variability Realization at the Architectural Level

At the code level, the variability realization of IAM systems requires to resolve process and topology variability. The upper part of Figure 12 shows the involved variability modeling constructs, as partially introduced in Figure 10. The lower part shows the variability realization mechanism in terms of code generation, in order to automate software derivation, which is one of the major goals of our work.

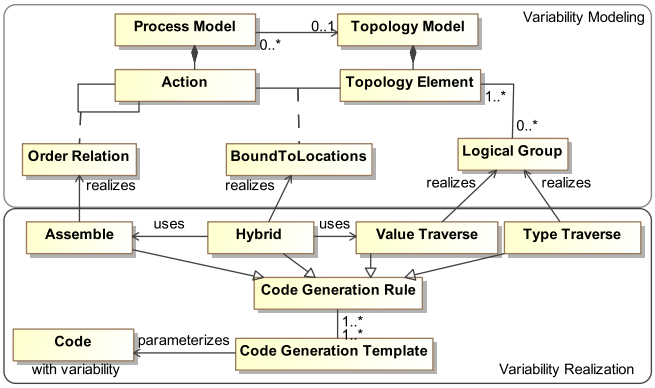

Figure 12: Variability Realization at the Code Level

As can be seen in Figure 12, we come up with three types of associations between elements of the multi-variability models. We explain and address the realization of each of them as follows:

- *Order relation:* The derivation stakeholders or modelers specify behavioral sequences in process models. The actions within the process models imply order relations [30]. In this context, the reusable artifacts are the individual implementations of each action. The realization of this relation requires to *assemble* the reusable code of actions according to their order in the process configurations. This assembly is implemented by generating the "glue-code" for invocation.
- *Bound-to-Locations relation:* This relation has been introduced already in Figure 10. The purpose of having it is to express that an action is to be executed at certain locations within the factories. The *hybrid* realization is necessary for this relation, based on both the assembly and value traversing.
- *Logical Group:* We introduce the *logical groups* in order to model a logical association between software services and their corresponding topological elements. The reason to have this association is that software services need to manage and observe different topological elements. Figure 13 shows two logical groups. For group 1, the two mini-load conveyors and two buffers belong to the logical group of the location tracking component, in which the transportation progress of boxes is monitored. For group 2, the component of buffer management coordinates only the two buffers in this example. The cardinality of topological elements within each logical group causes the variability. We realize this variability by *type traverse* or *value traverse* of the model instances within topology models.

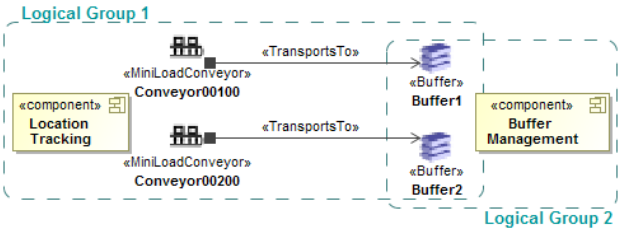

Figure 13: Examples of Logical Groups

3.3 A Semi-Automated Derivation Process

Based on the multi-variability modeling and realization, we establish the derivation approach shown in Figure 14. An initial version of this process has been reported in our previous study [13]. Our approach has three major phases: preparation, configuration, and generation. Preparation and generation are two (semi-)automated phases, aiming at saving the effort of derivation stakeholders. The configuration phase remains manual but tool-supported, in which the process variability, the topology variability, and their interrelations are configured by the three different roles of stakeholders according to their concerns (see Section 2.2). By doing so, we enable the user-centered variability modeling.

1.Preparation: This is a semi-automated activity. This step prepares a base configuration with reusable model templates and model fragments to help derivation stakeholders, so that they do not need to start a "construction" of a complex system from scratch.

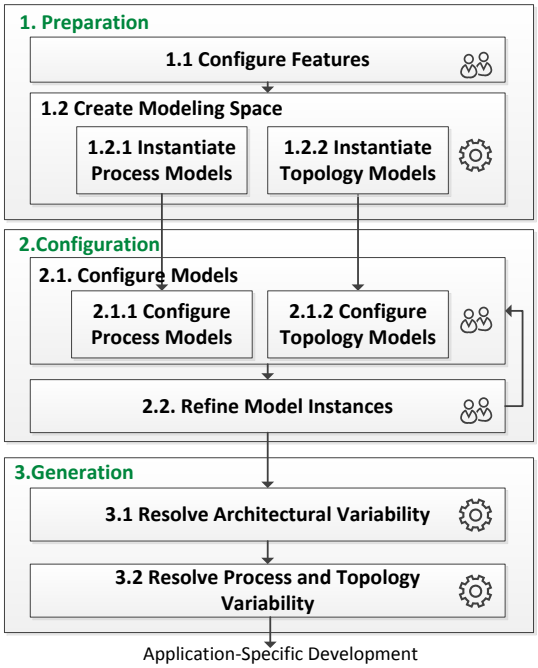

Figure 14: The Derivation Process

- **1.1:** As the first step, derivation stakeholders may already bind the feature-based variability, especially the global features in higher abstraction levels.
- **1.2:** The derivation infrastructure interprets the configured features with the associated components (see the association shown in Figure 10). Then automatically generates a modeling space, and instantiates the required process and topology model instances to prepare for further configuration.

2.Configuration: The manual configuration of different variability types is usually an iterative manner in our target domain, especially when process and topology variability are related in an intermingled way.

- **2.1:** The derivation stakeholders configure the instantiated process and topology models from Step 1.2.
- **2.1.1:** The requirement engineers configure the process model instances to specify the behaviors of the target systems.
- **2.1.2:** The hardware-oriented engineers configure the topology model instances to describe the configurations of the target systems.
- **2.2:** The software developers refine the configured process and topology models, associate the process elements with topological elements, add details, or correct errors when necessary.

3.Generation: Given the configured features, processes, and topology models, this derivation activity automatically generates the application-specific architecture and source code as (partial) products.

- **3.1:** The code generation engine realizes the variability at the architecture level, mainly by inclusion and exclusion of components. The output of this step is an intermediate work-flow of the final derivation step.
- **3.2:** The code generation engine uses the intermediate work-flow created in 3.1, and derives the application.

Since our target systems belong to a flexible category of software product lines, we expect that it is always necessary to have an application-specific development phase to finally fulfill all customer requirements. Software architects, developers, and testers potentially need to participate in the application-specific development.

4. CASE STUDY

The overall objective of conducting this case study is to understand the feasibility of apply the presented approach from a practical viewpoint, and to evaluate the improvement of it in a practical setting.

We have chosen a complex component of the warehouse management systems in Siemens as the subject of this case study, as it is a typical representative with regard to domain complexities to come to meaningful results. This chosen component has seven sub-components; two of them are optional components. According to the experiences of domain experts and analysis of legacy projects, the size of this component is usually around 10000 lines of code (LOC). In this software family, highly reusable utility functions, such as logging or data access, are handled already by the infrastructure of the software family. Currently, for each project, the code of this chosen component is completely hand-written.

This case study was planned according to [23], which contained the tasks of planning, executing, collecting data, and finally reporting the results. We plan to characterize and evaluate the outcome from the following three aspects:

- *Characterize the variability models:* Besides feature models, we introduce the two DSMLs for modeling processes and topology. We expect that by using multi-variability modeling and realization, more requirements of the component can be modeled and expressed. The aspired outcome would be to understand and characterize to which extent the chosen component depends on the three variability types.
- *Evaluate feasibility:* The approach should be applied in real-world contexts and settings. The effort of using the approach, especially during family engineering, should be affordable and reasonable.
- *Evaluate the automated level of derivation:* The approach provides a semi-automated software derivation process, supporting model instantiation and code generation. We expect that the approach improves the efficiency of derivation and development.

4.1 Execution Procedure

This sub-section describes the procedure of conducting the case study on the chosen subject. Following the family and application engineering processes in SPLE, our case study procedure has also two phases.

4.1.1 Family Engineering Procedure

Before we could start this case study, we have done domain analysis and language engineering, as introduced in Section 3.1. We developed the two domain-specific modeling editors on top of the modeling platform MagicDraw [1]. The implementation is based on UML profiles, in which meta-classes are defined as stereotypes. In addition, we have established the infrastructure of the derivation approach. The development was distributed over a 15-month time frame. Our approach aims at a broader scope of the domain of warehouse management systems. Especially, we validated the DSMLs in both Siemens-internal and external warehouse management systems with the participation of 7 domain experts. Our estimation of the effort spent of these works was approximately 7-10 person months.

Based on this foundation, one engineer experienced in model-based development and one architect of the warehouse management systems from Siemens participated in the execution of this case study. The procedure regarding the chosen component has three steps:

- *FE1.* Decide architecture: Several legacy projects of different customers have variants of this component. There are subtle structural differences, even though they were developed by the same team. We developed the reusable architecture based on the most mature component variant among them.
- *FE2.* Identify and locate variability in code: During the execution of this step, we identified and established the traces between variability (meta-)models and technical solution space. For example, we associated the optional sub-components with features. Within sub-components, we located the pieces of source code with the influential variability types, and added the corresponding models as parts of the core assets.
- *FE3.* Extract the reusable code assets: Classes without variability can be directly added to the core asset base. For classes with variability, a re-factoring step was necessary to separate common parts from variable parts. Variable implementation parts were, in simple cases, included via feature-like variability realization (cf. Section 3.2). For process and topology variability, code templates were extracted, following the realization patterns of assembling, value traversing, type traversing or hybrid composition (cf. Section 3.2).

To develop and extract reusable code assets at step F3, we applied several re-factoring patterns suggested in [18], such as hook methods, relocate classes, addition at the end of methods. We used Xpand as the template language to realize the variability in this case study [3].

Figure 15 presents an example of a code template to assemble classes with a hook method as the realization of the order relations for process variability. The template selects all actions of the type *PeriodicAction* from its targeted package. It generates a class for each action (line 7). Inside the generated classes, the $WakeUp$ and $NextStep$ methods separate the business logic of the current action and the glue code to invoke the next action. The reusable business logic is included via a dedicated template (at line 10). At line 17, the code template searches for outgoing edges of actions and generates the code to "wake up" the next one.

The template language supported by Xpand allows other combinations for filtering and iterating elements, with which we realize the bound-to-locations relations and logical groups. Finally, we added the developed reusable code templates as core assets for derivation.

4.1.2 Application Engineering Procedure

In application engineering, feature, process, and topology models are used to configure the systems and derive the products, according to the proposed derivation process in Section 3.3. An ideal validation would be to compare the development time with and without the proposed approach with entirely new customer requirements. As this is not practically feasible, we instead replayed the deriva-

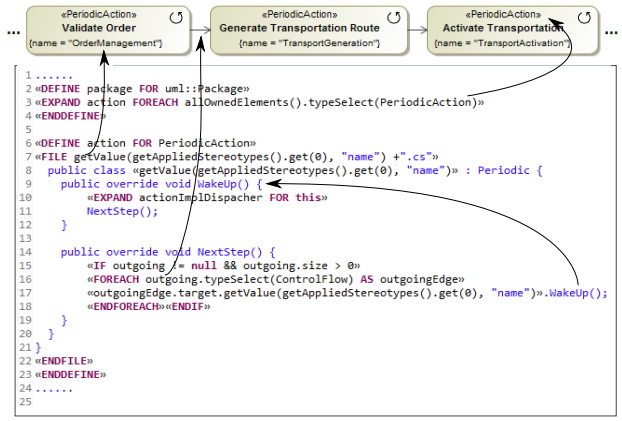

Figure 15: Realization of a Hook Method

tion of previously executed projects based on their original requirements. The derivation steps are as follows:

- *AE1.* Preparation: We configured features at the first place, and the derivation infrastructure instantiated the process and topology models that need to be further configured. The left-hand side of Figure 16 shows some examples of these models.
- *AE2.* Configuration: We used the developed process and topology editors to configure the instantiated models. Figure 16 also shows the screen-shots of the two editors with an example of binding an action to its locations of notification.
- *AE3.* Generation: The derivation infrastructure took all the configured models to generate the application-specific architecture and source code. We validated the generated code with the applicable unit tests from the original projects (with adaptations, where necessary) and with code reviews.

4.2 Reporting the Case Study

After finishing the procedure above, we collected data to gather evidence for characterization, evaluation, and discussion. This section reports on the results.

4.2.1 Characterize the variability models

Table 1 shows the identified reusable variability element types for the chosen complex component. We identified 16 features, including both architecture-relevant features that decide component inclusion and code-level features. For processes, 8 reusable process models with process templates were identified with a total of 31 reusable actions that can be instantiated. For topology, one topology model is needed for this chosen component that may comprise 8 different types of topology elements. Having these results, we can see that the two domain-specific modeling languages significantly enrich the modeling space of variability representation.

Table 1: The Number and Size of Variability Elements

	Feature	Process		Topology	
Number	16	Models	Actions	Models	Elements
		8	31	1	8

Among the seven sub-components, two of them are intensively process variable, and two of them are highly topological variable. The other three sub-components have more or less scattered types of variability. Some code parts comprise two or three influential variability types in a tangled way. Based on our experience, reducing the tangling of variability types for a piece of code makes extraction of code templates easier, while it increase the effort of re-factoring.

Discussion: We observe that the modeled topological elements in factories have different importance to software derivation. The *Stationary Elements* occur often in several different *logical groups*, which means that they are observed by or interact with software components more often than *Transporter Elements*. The reason is that their events are more likely to trigger processes or business logic. In contrast, *Transporter Elements* are usually tracked by the monitoring services, with less business logic.

4.2.2 Evaluate Feasibility

As reported in the procedure of family engineering (cf. Section 4.1.1), our case study is based on the foundation of the developed DSMLs, language editors, and the derivation infrastructure, which were developed aiming at a broader scope of the domain of warehouse management systems.

The time spent during family engineering for this particular case study was slightly less than 100 hours. This includes the activities of (F1) deciding the architecture, (F2) identify and locate variability, and (F3) extract code templates (see Section 4.1). The involved software architect reported that implementing such a component usually took 2 to 3 person months per project. He confirmed that the case study could show the feasibility of applying the approach in a practical setting. The experience was quite encouraging, which means that there is a good chance to further apply the approach to other complex components within this software family.

Discussion: Several factors have turned out to be critical to the feasibility of such an SPLE approach:

- Maturity of reference architecture: The decision making for coming up with the reusable component's architecture was rather quick in our case study. The effort was around 1 day of work. The reason is that the existing reference architecture of the component was already stable and quite satisfactory with the reuse purpose. The level of maturity of the component was critical in this regard. Each software family, each component, or even each artifact within the same systems, may come in different levels of maturity [6]. Improving the maturity of target systems or components may require significant effort, which will increase the required re-factoring work, and in the worst case, make the approach economically unfeasible.
- How mature is the domain: Bosch [6] points out that not all systems need to or can improve their maturity levels. It depends on how variable the domain is, and how good the variability models can capture the variability in both requirements and implementation concisely. Warehouse management is a relatively mature domain of Siemens business. The engineers working for this product family have accumulated good domain and technical knowledge.
- The accessible resources during the development of family engineering: It turned out that the participation of the architect from the warehouse management systems was critical to the decision making and variability mining steps. Besides, the expertise of modeling and code generation is also very important. This can lead to another risk, because not all development teams have such expertise on a regular basis.

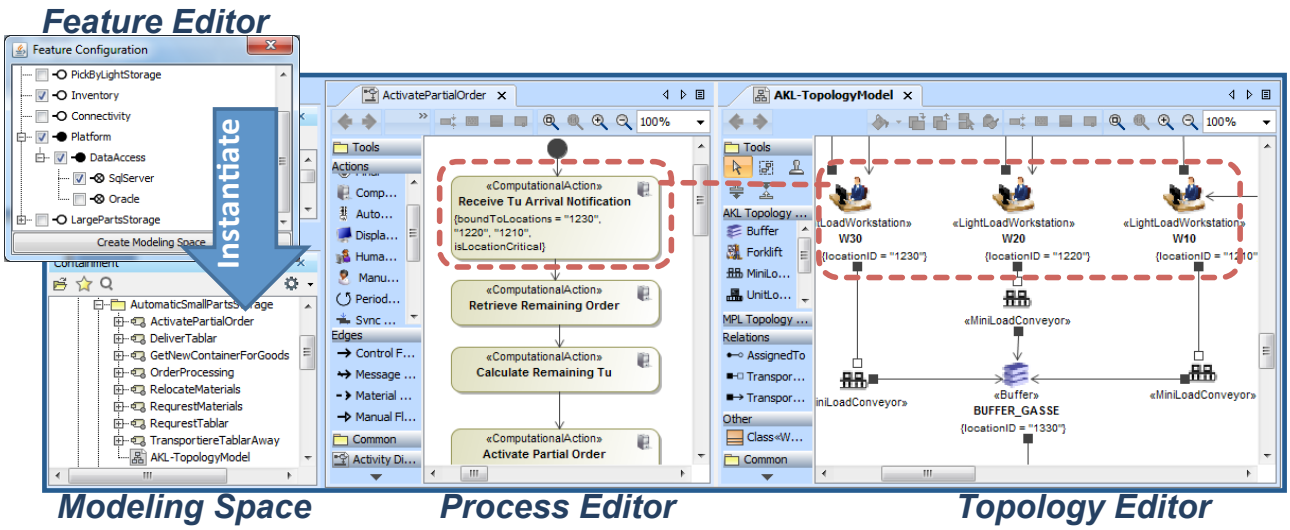

Figure 16: A Screenshot of the Developed Tool Support in MagicDraw

4.2.3 *Evaluate the Automated Level of Derivation*

Table 2 shows the types and size of the developed reusable asset base. The code-relevant assets in our context include Xpand templates for variable code and C# classes without variability. Besides, for the derivation infrastructure, we developed the required work-flows and script files for model instantiation and code generation (see the automated steps in Section 3.3), which serve as inputs of the derivation infrastructure supporting the automation.

Table 2: Reusable Asset Types and Sizes

Asset Types	Formats	LOC
Code with Variability	Xpand Code Template	5130
Directly Reusable Classes	C#	2399
Inputs to Infrastructure	MWE Work-flow and Builds	513
	Total	8042[1]

Following the case study procedure for application engineering, we selected the features, configured the instantiated processes and topology models, and derived code, according to the known requirements of two legacy projects. The goal is to get estimation about how good the automatic derivation and code generation can achieve. Table 3 shows the configured features, reused process actions, instantiated topological elements and automatically derived source code of the Project 1 (P1) and Project 2 (P2).

Table 3: Analysis of Automatically Derived Code

	Feature Config.	Process Actions	Topology Instances	Derived Project-Specific LOC
P1	16	29	51	5027
P2	12	24	67	4194

The original hand-written LOC of P1 and P2 are 11530 and 9869 respectively. Adding the common 2399 LOC shown in Table 2 to the derived project-specific LOC in Table 3, both of these two projects could reach a quite satisfactory automation level of code generation. While, the outcomes of the derived LOC for P1 and P2 show already a considerable difference of the automatically derived LOC. By analyzing the correlation of the size of configured and reused model elements, we see that the derivation may produce a wide range of LOC with the worst case around 1800 LOC. The size of automatically generated code depends on how "different" the requirements of the particular projects are, and how much glue code is needed. Furthermore, we understand that the result of this case study also includes significant subjective aspects. In particular, investing more effort on improving code templates or even restructuring the software architecture would have a significant impact on the outcome.

Discussion: We observe that components with topological variability can reach nearly full-generation, whereas the process variable components cannot reach the same automation level. The feedback from the involved architect reveals that this situation is actually aligned with his expectation. The realization of topological variability was based on type or value traversing, and these are well-supported in the template language. The realization of processes requires the assembly of reusable pieces of code in assets. Thus, for process variable artifacts, he expected to get more "glue-code" generated, especially when specified actions in processes cannot find reusable matching in assets. In these cases the developers have to implement the business logic manually.

4.3 Lessons Learned

Applying the approach enforces a more explicit reusable architecture. Firstly, we observe that the finalized reusable architecture has differences with the documented reference architecture of this chosen component. The documented reference architecture had not enough details to facilitate code generation. Secondly, the re-factoring enforces to collect variable code at certain places, e.g. in a code template, which "highlights" the non-variable code in the architecture. Because of this, we could find the directly reusable classes shown in Table 2, which were previous hand-written, although they were actually already reusable. Thirdly, we have also recognized that the developed architecture for reuse is not the optimized solution for reducing code complexity. The code generation templates in core assets have the goal of covering a significant amount of variants, so that to certain extent it has to sacrifice the understandability of the software architecture and code to gain generatability.

Using the DSMLs improves the willingness of modeling. DSMLs enable fine-grained associations between the requirements in problem space and the technical implementation in solution space. This becomes a motivation for "modelers"

[1]Notice that there are still four classes with very low reuse possibility, where we could only put some stub methods.

during manual configuration time to specify processes and topology with more effort and attention, reported by an architect. He pointed out that with such an approach the outcomes of modeling have impacts on the final systems. This is a significant improvement over the current state of practice, where requirement engineers specify the processes only informally and hand it over to hardware-oriented engineers and software engineers. The processes are only considered as input to be understood. Compared to this scenario, our approach actually encourages the stakeholders to produce better quality of models.

The highest possible rate of code generation should not be the goal. The initial goal of our case study was to strictly separate all generatable code from the hand-written code, to optimize the code generation rate. During re-factoring, it turned out that there are significant trade-offs to make. While there are several known patterns, such as hook methods, override mechanisms, or partial classes. They commonly only pay off for code blocks of significant size. Therefore, we only pursued to achieve reuse to a certain extent, avoiding too fine-grained reuse and to avoid increasing the complexities of the architecture.

Is the completeness of meta-elements important? The answer in our case depends on the purpose of modeling. At the DSML engineering phase, we tended to have much more topological supplementary meta-elements (cf. Figure 8). For example, there can be many different types of sensors, motors, etc. Soon we learned that not all real-world elements are necessary to be modeled for the purpose of software derivation in our targeted domain. When the primary goal is to represent the domain, more completeness is desirable. When the outcome of configured models serve as input "languages" for software derivation, it is important to limit the size and complexity of DSMLs. Not all information and entities resulting from real-world modeling is necessary for software derivation.

5. RELATED WORK

This section discusses the related works, focusing on approaches resolving multiple variability types. One of our previous works [13] has reported a previous version of the technical derivation process (see Section 3.3). This paper differs from the previous work, with the focus of variability modeling and realization, and it also reports on a case study of applying the approach in practical settings.

Many existing works propose approaches for topology or process variability. For modeling topology variability, Behjati et al. propose SimPL [4] as a methodology that provides multiple views to the users to specify integrated controls systems (ICS). In SimPL, the variability of software, hardware, and dependencies among them are presented in different views. Urli et al. propose SpineFM [27], which is a tool-supported approach that interrelates domain models with feature models. SpineFM enables an order-free configuration process by propagation mechanisms to help users to perform their tasks. Berger et al. distinguish feature- or decision-like variability and topological variability, and report on the applicability and challenges of modeling topological variability with a case study in the domain of fire alarm systems [5]. Dhungana et al. propose to use domain modeling to express hierarchies and multiplicity of variability product families [11]. We have been inspired by these approaches, for example by SimPL and SpineFM. None of these approaches, however, address explicitly feature, topology, and process variability, and integrate them with a complex derivation process, as it is common in IAM systems.

There are also existing works aiming at resolving process variability. For instance, Gröner et al. proposed an approach towards the configuration and validation of business process families, in which they associate feature models, process model templates, and mappings among the elements in these two types of models [14]. The business processes have usually execution semantics that can run directly on a work-flow engine. However, business processes do not address the concerns of material flows in the IAM domain. Without an extension of existing notations, it is not possible to express the IAM-specific variability, for example the bound-to-locations relations between process and locations. Brown et al. propose to attach behavioral models to feature nodes to compensate the representation limitation of feature modeling [7]. Heinzemann and Becker propose to use adapted concepts of the UML, named MechatronicUML, to enable hierarchical re-configuration of complex software components for self-adaptive mechatronics systems [16]. Pillai et al. propose a model-based approach applied to paper handling in printers for control software generation [21]. Trask et al. combine models to SPLE in the radio domain [26]. The limitation of the above approaches for the IAM domain is that topology variability is not of their concerns.

In addtion, mechanisms for constraints and consistency checking among several model types have not yet integrated into our approach. Kowal and Schaefer propose an incremental consistency checking approach. They point out that the consistency checking has several scopes and levels [17]. Nie et al. propose the automation support for product configuration in the context of cyber physical systems [20].

6. CONCLUSION AN FUTURE WORK

We present a model-based approach to address the complex variability of information systems in the industrial automation domain. The approach has three major parts. Firstly, the approach involves feature models to represent the common variability types, and two domain-specific modeling languages to express process and topology variability. Secondly, the three types of variability models are associated with the reusable architectural and code artifacts, to get prepared for software derivation. Thirdly, a semi-automated derivation process integrates the configuration of feature modeling and the two domain-specific modeling languages to improve on derivation efficiency.

We report on a case study by applying our approach on a chosen complex component within a software family of warehouse management systems in Siemens. The results show that using domain-specific modeling languages can improve on expression power of variability representation. By configuring the feature, process, and topology models, the derivation and generation can also reach a satisfactory level of automation. The feasibility of the presented approach is demonstrated within the setting of the case study. As a part of the contributions, we reported our lessons learned.

As future work, we intend to expand the case study to other sub-domains of IAM systems, to further evaluate the generalizability of the proposed approach. We also plan to interview domain experts to understand their perceptions regarding the potential approach adoption.

7. REFERENCES

[1] MagicDraw. http://www.nomagic.com/products/magicdraw.html.

[2] MOF. http://www.omg.org/spec/MOF/2.5/PDF/.

[3] Xpand. http://wiki.eclipse.org/Xpand.

[4] R. Behjati, T. Yue, L. Briand, and B. Selic. Simpl: a product-line modeling methodology for families of integrated control systems. *Information and Software Technology*, 55(3):607–629, 2013.

[5] T. Berger, Ş. Stănciulescu, O. Øgård, Ø. Haugen, B. Larsen, and A. Wasowski. To connect or not to connect: experiences from modeling topological variability. In *Software Product Line Conference (SPLC)*, pages 330–339. ACM, 2014.

[6] J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Software Product Lines Conference (SPLC)*, pages 257–271. Springer, 2002.

[7] T. J. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan. Weaving behavior into feature models for embedded system families. In *Software Product Line Conference (SPLC)*, pages 52–61. IEEE, 2006.

[8] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Software Product Line Conference (SPLC)*, pages 81–90. Carnegie Mellon University, 2009.

[9] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143, 2011.

[10] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *workshop on variability modeling of software-intensive systems*, pages 173–182. ACM, 2012.

[11] D. Dhungana, H. Schreiner, M. Lehofer, M. Vierhauser, R. Rabiser, and P. Grünbacher. Modeling multiplicity and hierarchy in product line architectures: Extending a decision-oriented approach. In *2014 IEEE/IFIP Conference on Software Architecture (WICSA)*, page 11. ACM, 2014.

[12] M. Fang, G. Leyh, C. Elsner, and J. Doerr. Experiences during extraction of variability models for warehouse management systems. In *Software Engineering Conference (APSEC)*, pages 111–116. IEEE, 2013.

[13] M. Fang, G. Leyh, C. Elsner, J. Doerr, and J. Zhao. Towards model-based derivation of systems in the industrial automation domain. In *Software Product Line Conference (SPLC)*. ACM, 2015.

[14] G. Gröner, M. BošKović, F. S. Parreiras, and D. GašEvić. Modeling and validation of business process families. *Information Systems*, 38(5):709–726, 2013.

[15] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, 2003.

[16] C. Heinzemann and S. Becker. Executing reconfigurations in hierarchical component architectures. In *ACM Sigsoft symposium on Component-based software engineering (CBSE)*, pages 3–12. ACM, July 2013.

[17] M. Kowal and I. Schaefer. Incremental consistency checking in delta-oriented uml-models for automation systems. In *Workshop on Formal Methods and Analysis in Software Product Line Engineering*, 2016.

[18] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed. From requirements to features: An exploratory study of feature-oriented refactoring. In *Software Product Line Conference (SPLC)*, pages 181–190. IEEE, 2011.

[19] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *Workshop on Domain-Specific Modeling*, 2004.

[20] K. Nie, T. Yue, S. Ali, L. Zhang, and Z. Fan. Constraints: The core of supporting automated product configuration of cyber-physical systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 370–387. Springer, 2013.

[21] C. Pillai, R. Fabel, and L. Somers. *Model Based Control Software Synthesis for Paper Handling in Printers*. Eindhoven University of Technology, 2009.

[22] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52(3):324–346, 2010.

[23] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[24] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[25] J.-P. Tolvanen and S. Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *Software Produce Line Concerence (SPLC)*, pages 198–209. Springer, 2005.

[26] B. Trask, D. Paniscotti, A. Roman, and V. Bhanot. Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In *Software Product Line Conference (SPLC)*, pages 846–853. ACM, 2006.

[27] S. Urli, M. Blay-Fornarino, and P. Collet. Handling complex configurations in software product lines: a tooled approach. In *Software Product Line Conference (SPLC)*, pages 112–121. ACM, 2014.

[28] A. Vallecillo. On the combination of domain specific modeling languages. In *European Conference on Modelling Foundations and Applications (ECMFA)*, pages 305–320. Springer, 2010.

[29] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Line Conference*, pages 233–242. IEEE, 2007.

[30] M. Weidlich, J. Mendling, and M. Weske. A foundational approach for managing process variability. In *Advanced Information Systems Engineering*, pages 267–282. Springer, 2011.