Developing Model-Driven Software Product Lines

Doctoral Dissertation by

Xiaorui Zhang

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree Philosophiae Doctor (PhD) in Computer Science

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1463

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

This thesis focuses on model-driven software product line development, which is the combination of the following two software development paradigms: (1) Model-Driven Engineering (MDE), which focuses on modeling software products and automating code generation from product models. In particular, Domain-Specific Modeling (DSM), as a technique in the arena of MDE, is about defining a Domain-Specific Language (DSL) and creating software product models using the language. (2) Software Product Line Engineering (SPLE), is a means to produce similar software products, by consolidating those into product lines to enable managed reuse. In a model-driven Software Product Line (SPL) which adopts DSM technique, products are represented as product models defined in a DSL. The variability (and commonality) of all intended products is specified in a product line model, typically using a variability modeling language. Based on the variability specified in the product line model, reusable model fragments specified using the base DSL, serving as the core assets of the product line, will be reused to derive all intended product models. This thesis provides methods for developing model-driven software product lines, in terms of development methodology, automated assistance and SPL evolution support.

Firstly, this thesis presents two results on the methodology for developing a model-driven SPL: (1) A generic and separate variability modeling language, which can be used to specify a product line model defining how intended product models can vary from each other, both at the domain conceptual level and the realization level (model object level). (2) Guidelines on how to define a DSL that is suitable to serve as the base language for a model-driven SPL, if the base language of the product line does not exist yet.

Secondly, this thesis reports on two results in providing automated tool support for model-driven product line development: (1) A method for synthesizing a product line model from a set of existing product models when the product line is not built from scratch. (2) A method for ensuring that all the product models that can be derived from the product line model are intended.

Thirdly, this thesis reports on three results in providing support for evolving model-driven SPLs: (1) A method for augmenting the existing product line model when new product models need to be included. (2) A method for suggesting automatic update to the product line model when the core assets of the product line have been changed. (3) A method for calculating semantic difference between two model-driven SPLs.

We illustrate the application of our approaches in various case studies in different domains, provided by both industry and academia. Different phases of SPL development and evolution can require substantial amount of manual efforts, of which productivity can be improved by adopting our automatic tool support. We show that

by following our approaches, model-driven SPLs can be developed and evolved in a systematic and efficient manner.

To Andreas, thank you for accompanying me to the park in that sunny afternoon, where everything started.

Acknowledgement

The work presented in this thesis has been performed in the context of the MoSiS project (Model-driven development of highly configurable embedded Software-intensive Systems), and has been funded by the Norwegian Research Counsil (project number 180110/I40).

First of all I would like to thank my two supervisors Øystein Haugen and Birger Møller-Pedersen for their invaluable guidance through my entire PhD process. I would also like to extend my gratitude toward the other researchers in the MoSiS project at SINTEF, Andreas Svendsen, Franck Fleurey and Gøran K. Olsen, for their support and collaboration.

I would like to thank Bjørn Skjellaug, my research director at SINTEF ICT, for giving me the opportunity to write this thesis. In addition I thank my other colleagues at SINTEF for providing the most pleasant, inspiring and professional work environment.

I really appreciate the cooperation with ABB Corporate Research and Agresso, for their collaboration in problem identification, prototype design and evaluation. Their willingness to try out model-driven software product line development has been an inspiration for my work.

I would like to thank my in-laws, Kari, Rolf, Lisa, Anne-Elisabeth and Arne, for giving me encouragement, love and sometimes free dinners ☺ during my PhD study. I am also very grateful to my friends, Yun, Jie and Haowen, for supporting me all the way in terms of both encouragement and good home-made food ☺

I would like to express my deep gratitude to my parents and (late) grandparents, for being good people that I can look up to, and for their guidance during my personal development.

Most of all, I am sincerely grateful to Andreas Svendsen, who has been my colleague, my husband and my best friend over the years. We worked together, travel together and live together. When I was sad, he cheered me up; when I was happy, he felt happy for me; when I wanted to give up, he encouraged me; when I prolonged my PhD without funding, he supported me. Without him, this thesis would never be made possible.

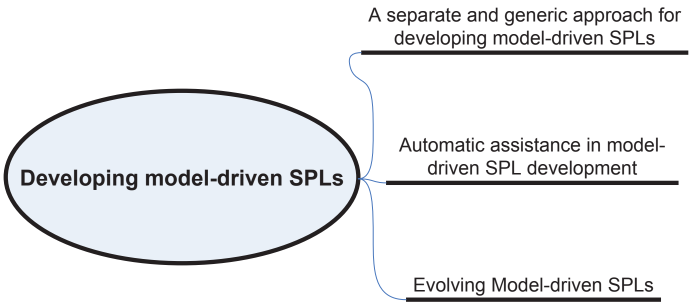
Table of Contents

1 Introduction

A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [40]. Software Product Line Engineering (SPLE), emerging as a viable software development paradigm, enables the reduction of time-to-market for similar software products through managed reuse of core assets.

As another software development paradigm, Model-Driven Engineering (MDE) focuses on creating software models and automating code generation from the models [42]. Software models can be specified using either general-purpose modeling languages (e.g., UML) or Domain-Specific Languages (DSLs). A DSL is a custom-made language for a specific domain [76]. Typically the language constructs and rules of a DSL only capture the essential concepts of the domain. Therefore, a DSL allows domain experts to model systems using familiar domain-specific terms, without having extensive modeling experience.

Model-driven SPL development combines the SPLE and MDE paradigm. In model-driven SPLs, core assets are reusable software model fragments instead of reusable code snippets. Therefore, products of model-driven SPLs are in the form of models (we call them "product models"), from which the code for the software products can be further generated through model-to-text transformations.



Developing model-driven SPLs
A separate and generic approach for developing model-driven SPLs
Automatic assistance in model-driven SPL development
Evolving Model-driven SPLs

Fig.1. Thesis contribution overview

As illustrated in Fig.1, the objective of our thesis is to provide methods to facilitate effective and efficient development of model-driven SPLs. In particular, we address how to develop model-driven SPLs for different domains in a separate and generic way, how to improve the productivity of model-driven SPL development by automatic means, and how to improve the productivity in evolving model-driven SPLs.

In a model-driven SPL, instead of creating similar software product models individually, the product models are derived from the product line model. The product line model, often in the form of a variability model, is created to specify the variability (and commonality) of all the intended product models. There are two strategies for specifying a product line model [66]:

(1) The amalgamated approach is to extend the base language (either a general-purpose modeling language or a DSL) of the product line with variability modeling concepts. However, changing the definition of the base language and its tool support (e.g., editors and code generators) to facilitate variability modeling may not always be feasible.

(2) The separate approach is to describe the variability of the product line using a dedicated variability modeling language. For example, feature modeling techniques, first proposed by Kang et al. [71] for domain analysis purposes, belongs to this category. In a feature model (product line model), the variability (and commonality) of the product line are represented as features that are hierarchically organized. In order to derive a product model, the developer does not only need to choose all the required features from the product line model, but also need to define feature/variability realization - how the chosen features should be realized by reusing the core assets (reusable model fragments) during product derivation (model-to-model transformations) [14].

However, including feature modeling, most of the separate variability modeling techniques [71, 99] do not include language concepts to specify how features should be realized at the (model) object level. Furthermore, it is a challenge to define feature realization in a separate and generic way for product lines in various domains with product models specified using different base DSLs.

In this thesis we provide a separate and generic approach for developing model-driven SPLs (see Section 6.1.1), which allows the developer to define both features and their realizations holistically in a product line model.

When a DSL is chosen to be the base language for a model-driven SPL, all the core assets and intended products will be specified in this language. Moreover, if this DSL does not exist yet, to create it will become one of the prerequisites prior to the actual SPL development.

In this thesis we report on experience in developing a base DSL that is suited for building model-driven SPLs afterwards (see Section 6.1.2). We show that a properly defined DSL, together with well-planned SPLs, can improve the productivity for developing software products [144].

Providing automatic assistance to model-driven SPL development can increase productivity in the production of software products beyond current human labor levels. Numerous automatic tools have been developed to support model-driven SPL development. However, there is still a lack of automatic assistance for many specific needs in various contexts at each development phase.

In this thesis we provide a set of automatic methods to improve the productivity in identifying variability (see Section 6.2.1) and defining variability/feature realization (see Section 6.2.2) for SPLs.

Software product lines often evolve over time [120]. Many existing automatic techniques in SPL evolution have their focus on managing and understanding product line evolutions (e.g., version control systems and differencing tools [6]). Very few of them target on suggesting evolution steps automatically based on new requirements, such as augmenting an SPL with new products [141].

In addition, there is also a lack of automatic methods in supporting SPL co-evolution, such as suggesting necessary update to the product line model after the core assets have been changed.

Furthermore, most of the differencing tools being used to understand the impact of an SPL evolution are either syntax-based, which has its limitation in revealing added/removed products during an evolution, or semantic-based which only compares two SPLs at the feature/variability specification level without considering the feature/variability realization that might have been changed.

In this thesis we provide a set of automatic methods to improve the productivity in SPL evolution, in terms of augmenting an SPL with new products (see Section 6.3.1), co-evolving the product line model when the core assets are changed (see Section 6.3.2), and differencing two SPLs semantically by taking both features and their realizations into account (see Section 6.3.3).

This thesis work has been performed in the context of the MoSiS[1] project. MoSiS is an industrial-driven research project with focus on developing and standardizing a generic variability modeling language, as well as promoting the model-driven SPL paradigm to industry.

There is no silver bullet for software engineering problems [32]. Thus, rather than searching for the silver bullet for model-driven SPL development, in this thesis we make our research efforts in contributing to a technology box with specialized tools and methods tailored for specific needs.

1.1 Overview of the Contributions

As illustrated in Fig.1, our work on developing model-driven SPLs is addressed through the following areas:

- A separate and generic approach for developing model-driven SPLs in different domains.
- Automatic assistance in model-driven SPL development.
- Evolving model-driven SPLs.

In the following we give a detailed description of the challenges in these areas.

1.1.1 A Generic Approach for Developing Model-Driven SPLs

Defining variability specification and realization in a generic way (see Section 6.1.1). There are two challenges that we address in proposing a separate and generic approach for developing model-driven SPLs:

[1] http://www.itea2.org/project/index/view?project=200

(1) One challenge is that, few separate and generic model-driven approaches (e.g., feature modeling) support defining both the domain-level variability (features) and their realizations at the model (object) level holistically in the same product line model.

(2) The other challenge is how to represent feature/variability realization (model editing operations) in a generic way. The approach should provide means to describe arbitrary edits to any model specified in any base DSL.

In order to address these challenges, we propose a separate and generic variability modeling language, the Common Variability Language (CVL) and the CVL methodology for SPL development. The CVL language provides capabilities in defining both variability/feature specification and realization in the same product line model. Furthermore, the CVL language categorizes arbitrary model edits into value, reference and fragment substitutions, which can describe any value and structural changes in any model specified in any MOF-based modeling language. Since the CVL language is the core of this approach, we refer to this approach as "CVL" in the rest of this thesis.

Defining a Base DSL that is Suitable for Building Model-Driven SPLs (see Section 6.1.2). DSM/MDE and model-driven SPL are both new paradigms for most developers in industry. Very often a base DSL needs to be developed together with model-driven SPLs, which raises the question: how to develop a base DSL that is suitable for building model-driven SPLs that are based on separate variability modeling approaches?

We report our experience in developing a base DSL and SPLs for the payroll reporting domain. We show that: (1) If the language concepts of the base DSL is fully domain-specific without any variability modeling concepts, it will be more intuitive and conceptually clearer to build SPLs that are based on separate variability modeling approaches. (2) How the productivity of traditional software development can be improved by model-driven SPL techniques. Since the experience was collected during the development of the Agresso Payroll Reporting Language (APRiL), we refer to this contribution as "APRiL" in the rest of our thesis

1.1.2 Automatic Assistance in Model-Driven SPL Development

Synthesizing an SPL from a set of existing products (see Section 6.2.1). As a new software development paradigm, SPLE is not always adopted from scratch in practice. For example, when an organization shifts from traditional software development to product line development, the developer often needs to include existing products in a product line and further enhance it to introduce new products [142]. For product line development in this context, we see the potential in providing automatic assistance to identify variability (and commonality) of an SPL. We show in this thesis how to synthesize a set of existing product models into a preliminary product line model specified in our generic variability modeling language, through an automated procedure. This preliminary product line model can serve as the base line for manual enhancement. Since this approach is built based on CVL and model comparison techniques, we refer to this approach as "CVL Compare" in the rest of this thesis.

Ensuring that the variability realization will only yield intended products (see Section 6.2.2). In order to derive product models from a product line model, the developer does not only need to specify the variability of the product line, but also needs to define how the variability (features) can be realized by reusing the core assets (reusable model fragments) and applying necessary model editing operations. However, specifying variability/feature realization is often an error-prone process since it requires the developer to have a good understanding of both the core assets and intended product models at the model object level. In particular, there are two challenges that we focus on in this thesis:

(1) With most existing techniques, the developer does not have immediate feedback on his/her specification changes to the variability/feature realization at design time. Therefore the more complex the variability/feature realization is at the model object level, the more difficult it will be for the developer, without proper tool support, to ensure that the current definition of the variability realization will only yield intended products.

We address this challenge by providing a generic variability realization simulator, which can be evoked at design time to simulate the execution of the variability realization and provide a preview of the resulting model excerpt. The simulator, if properly used in an iterative "define-preview-improve" manner, can provide an immediate feedback on whether the current definition of the variability realization will yield intended model changes in the final product models.

(2) Most variability/feature modeling techniques provide means to specify domain-level constraints that govern dependencies between features [22, 45]. For example, feature A implies feature B, indicates that these two features should always coexist in a product. However, if both the realizations of feature A and B require to change the same model object/reference, but in two different ways, then an inconsistency between the feature specification level and the feature realization level occurs. During the derivation of products with feature A and B, this inconsistency can cause errors because the realizations of feature A and B contradict with each other.

We address this challenge by categorizing such inconsistencies and proposing a consistency checker to search for unwanted inconsistencies that may halt the product derivation or yield unintended products. In the rest of the thesis, we refer to this approach as "Automatic assistance in defining variability realization" in the rest of this thesis.

1.1.3 Evolving Model-Driven SPLs

Augmenting an SPL with new products (see Section 6.3.1). Product lines are often subject to changes over time [120]. Augmenting an existing product line to include new products is a typical product line evolution scenario in practice and it has been so far mostly a manual process [141]. This process does not only require the developer to perform an extensive comparison of the new and existing products, but also to have a comprehensive understanding of the impact of each change to the existing product line.

In this thesis we show how a product line model specified in the Common Variability Language (CVL), which is our generic variability modeling language, can be augmented with new product models, through a series of automatic routines, resulting in a tentative augmented product line model for manual enhancement [141]. We refer to this approach as "Augmenting an SPL" in the rest of this thesis.

Co-evolving the product line model when the base model is changed (see Section 6.3.2). For a model-driven SPL, all its product models can be derived by reusing and changing the reusable model fragments (core assets of the SPL). However, core assets can undergo maintenance for various reasons. Therefore a series of questions arise: Will the product line model still derive the intended product models from the core assets that have been changed? If not, how the product line model should be updated to ensure that the product derivation remains unaffected?

We address this challenge by proposing an approach for co-evolving the product line model (developed in our CVL language) when the base model (part of the core assets) is evolved. In particular, the approach detects the inconsistencies in the original product line model caused by the changes to the base model, and automatically suggests an evolved product line model which has all the inconsistencies resolved. We refer to this approach as "Co-evolving an SPL" in the rest of this thesis.

Semantic Differencing for SPLs (see Section 6.3.3). In order to understand the impact of an SPL evolution which has taken place over time, it is common for the developer to compare the original and the evolved product line. When it comes to applicable differencing techniques, syntax-based approaches have their limitations in situations when syntactical similar models have very different semantics, which has been observed in feature models [6]. It would be helpful for the developer to gain an understanding of the semantic impact of an SPL evolution (e.g., in terms of added/removed products). Nevertheless, existing semantic differencing techniques for feature models only compare domain-level features without taking the actual feature realizations into consideration, resulting in an incomplete picture of the SPL evolution.

We address this challenge by proposing an approach for semantic differencing for SPLs. The approach is built based on the definitions of two semantic differencing operators, which take both feature/variability specification and feature realization into account during the SPL differencing process. We refer to this approach "Semantic Differencing for SPLs" in the rest of the thesis.

1.2 Structure of the Thesis

This thesis is presented as a collection of research papers with an accompanying overview. It is divided into two parts: Part I contains the overview, which gives the motivation, background and overview of the contributions. Part II is the main contribution in the form of a set of papers.

In addition to the introductory chapter, the remainder of Part I is organized as follows:

- In Chapter 2, we give an overview of the background of the thesis work.
- In Chapter 3, we elaborate the problem area and define research topics investigated in the thesis work.

- In Chapter 4, we describe the research methods applied in the course of the thesis work.
- In Chapter 5, we give a review of the literature and state-of-the-art.
- In Chapter 6, we give an overview of our contributions and research papers.
- In Chapter 7, we discuss and evaluate the accomplished work towards the research topics.
- In Chapter 8, we conclude and propose some directions for future work.

Part II contains seven papers in Appendix I - VII, which define the main contribution of the thesis.

2 Background

In this section, we give an introduction on the definition of the research topics that this thesis covers.

2.1 Model-Driven Engineering and Domain-Specific Modeling

Model-Driven Engineering (MDE) raises the abstraction level of typical software development, by shifting the focus from programming to modeling and automating code generation from the models. Software models can be specified using either general-purpose modeling languages (e.g., UML or Domain-Specific Languages (DSLs)). Domain-Specific Modeling (DSM) techniques allow domain experts to develop software applications of the domain without having extensive modeling and programming experience. For example, instead of coding a software system directly, domain experts can specify models of the system using domain-specific language concepts provided by a DSL. Subsequently, the models can be transformed into the code of the system by automatic code generators.

For example, as reported in [121], Train Control Language (TCL) is a DSL for specifying train control systems equipped at train stations. TCL with its tool support (i.e., TCL graphical editor and TCL code generator) is developed by SINTEF[2] in cooperation with ABB, Norway[3]. Traditionally train control experts at ABB need to develop train control systems for different station drawings received from the national railway authority. Such system development involves much coding for Programmable Logic Circuits (PLCs) using low-level programming languages, which can often be an error-prone and time-consuming process. TCL was developed to address this challenge. With the TCL graphical editor, train control experts can specify station models using the language constructs that graphically resemble the building blocks in the station drawings. Code for on-station PLCs can be generated from TCL station models through the TCL code generator.

Applying DSM techniques can improve the productivity in developing domain-specific software applications. However, the improvement also comes with an overhead, including the development of the DSL itself, DSL model editors and code generators.

[2] http://www.sintef.no/
[3] http://new.abb.com/no

There are two popular approaches for developing DSLs: (1) Extending the standard UML language with domain-specific concepts using UML profiles[4]. (2) Creating DSLs from scratch using metamodeling techniques. Our thesis work focuses on the latter approach.

Defining a DSL using metamodeling techniques includes three parts:

Abstract syntax, which is a set of rules about how language concepts can be used during the modeling process, defined in a metamodel. A DSL editor enforces the abstract syntax of the language so that only models conforming to the metamodel are allowed in the editor. As illustrated in Fig.2, in a TCL station model *LineSegment(s)* and *Switch(es)* can only be connected by *Endpoint(s)*, which conforms to the abstract syntax of the TCL language.

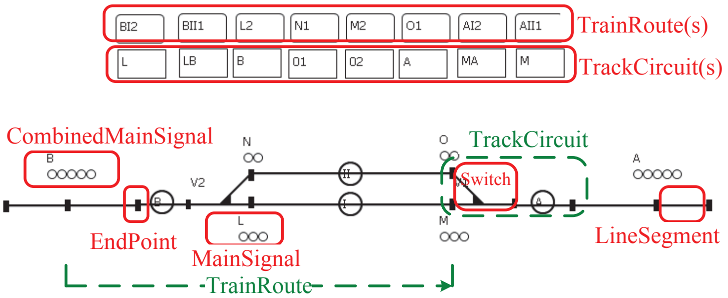


Fig.2. Basic TCL concepts in the graphical editor with annotations

Concrete syntax, which is a set of rules that define the way models look like to the modeler (domain expert), i.e. textual/graphical notation of the language concepts. It is important that the concrete syntax of a DSL resembles the counterpart of the domain visually, so that it may be easier for domain experts to comprehend the notations of this DSL. A DSL editor, either textual or graphical, is built based on the concrete syntax of the language. As illustrated in Fig.2, the concrete syntax of the TCL language is very domain-specific, which resembles the look-and-feel of the station drawings received from the authority.

Semantics, define what language concepts (and compositions of language concepts) mean, making it possible to understand models specified in this language precisely. As illustrated in Fig.2, the round-angled rectangles and the square-angled rectangles represent *TrainRoute(s)* and *TrackCircuit(s)* respectively. The semantics of a *TrainRoute*is a route between two *MainSignal(s)* in the same direction. The semantics of a *TrackCircuit* is the shortest segment where the presence of a train can be detected.

There are several tools for metamodeling DSLs and building DSL editors (e.g., Eclipse Modeling Framework core (EMF core)[5] for creating metamodels, Graphical

4 http://www.omg.org/spec/
5 http://www.eclipse.org/modeling/emf/

Modeling framework (GMF)[6] for building graphical editors and EMFText[7] for building textual editors). Take the TCL language for example, the TCL metamodel is developed using EMF, and the TCL graphical editor is developed using GMF. We also use EMF and GMF in the prototype development of our thesis work, which will be elaborated later in Section 6.

Code generators are responsible for transforming models specified in the DSL editor into code. In particular, a code generator is written as a transformation script, which reads in models, traverses model elements and transforms models into texts. Code generators can be developed using general-purpose programming languages (e.g., Java) or model transformation tools (e.g., QVT[8] and MOFScript[9]). We use MOFScript, a tool for model-to-text transformation, to develop the code generators in our thesis work.

2.2 Variability Modeling and Model-Driven SPL Development

Developing similar software products is a common software development scenario in practice. For example, in mobile phone industry, software systems for different phone models are quite similar to each other, since all of them need to support mandatory features such as calling and SMS. However, they also vary from each other by supporting different optional features. For example, a higher-end phone may be equipped with features like GPS, camera while a lower-end one may only have camera but not GPS. In order to reduce time-to-market when developing similar software products, ad-hoc code reuse (e.g., copy & paste) is often applied by developers. However, unplanned and unmanaged code reuse can introduce potential errors into the code and does not always maximize the benefits of reuse. In order to address these challenges, Software Product Line Engineering (SPLE) has been introduced to enable planned and managed reuse in the development of similar software products. Instead of developing similar software products individually, SPLE paradigm focuses on building a Software Product Line (SPL) from them. An SPL captures the variability and commonality of all its intended products. A set of core assets (reusable artifacts, such as code libraries, software components and etc.) serve as the base for an SPL, which will be reused to derive all intended products.

Model-driven SPL development is a combined paradigm of MDE and SPLE. In a model-driven software product line, core assets are reusable model fragments specified in a base language (e.g., UML or a DSL) instead of actual code snippets. All products are represented as models specified in the base language as well. The development of a model-driven SPL consists of the following phases:

Variability Identification. This phase focuses on capturing the variability and commonality of all intended product models of the product line. Variability identification has been mostly a manual process and the majority of contributing

[6] http://www.eclipse.org/modeling/gmp/
[7] http://www.emftext.org/index.php/EMFText
[8] http://www.omg.org/spec/QVT/1.1/
[9] http://marketplace.eclipse.org/content/mofscript-model-transformation-tool#.UpTAcdJIJvA

methods are directive guidelines. As the first domain analysis methodology, FODA [71] suggests to identify the variability of a domain by conducting surveys/interviews towards domain experts/end-users, as well as inspecting relevant documents and applications. Many other research works suggest similar methods for variability identification in SPL development, such as in FORM [72], FAST [132], PuLSE [23] and KobrA [18].

Variability Specification. In this phase, the developer specifies a product line model to describe the variability and commonality of the product line. There are two strategies to specify a product line model:

The amalgamated approach, which is to extend the base language (e.g., UML or a DSL) with variability modeling language concepts. However, it may not be always feasible to change the definition of the base language and its tool support (editors, code generators and etc.). Or the developer may prefer to keep the scope of the base language more domain-specific without offering variability modeling capability. Furthermore, with the amalgamated approach, the developer needs to repeat the work of extending the base language with variability modeling concepts when he/she starts building a product line with a new base language.

The separate approach, which is to specify the variability of a product line in a separate variability model using a generic variability modeling language. The variability modeling language is defined beyond the base language of the product line.

Feature modeling, first proposed by Kang [71], has been widely used to specify product line models. In feature modeling, a "feature" is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [71]. The variability and commonality of a product line can be represented as hierarchically organized features in a feature model.

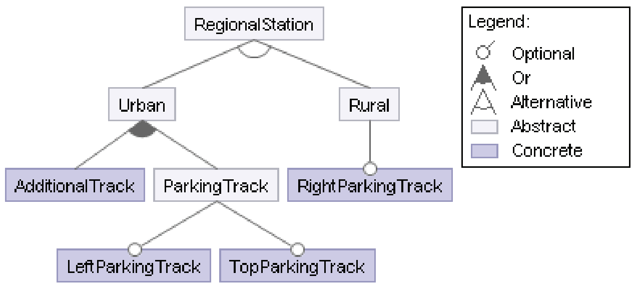


Fig.3. Feature model of the train control product line specified using FeatureIDE

Variability Realization. In order to develop an executable product line, it is not adequate to only identify the variability (features) of the product line and specify it in a product line model. In addition, the developer also needs to define how features/domain-level variability should be realized at the model (object) level by reusing the core assets (reusable model fragments) of the product line. For example, in order to realize a specific feature, it may be necessary to edit a specific model fragment (part of the core assets) slightly, assemble several model fragments together or take away some part from a model fragment.

Going back to the train control example: The train control experts see that many station drawings that they receive from the authority are very similar. Therefore they decide to develop train control product lines instead of specifying every TCL station model individually [123]. Fig.3 shows the feature model of a train control product line, which specifies the domain-level variability (features) of the product line using FeatureIDE, which is a popular feature diagram editor [74]. As shown in Fig.3, stations are categorized into *Urban* and *Rural* ones depending on their location. Urban stations can have one *AdditionalTrack* compared to rural stations. Urban stations can also have a *LeftParkingTrack* and/or a *TopParkingTrack*. Rural stations can choose to have an optional *RightParkingTrack*.

As illustrated in Fig.3, features are distinguished as abstract and concrete features. Thüm et al. [125] define abstract features as those that are "only used to structure the model and selecting or eliminating them does not make any difference in the generated variant code". As in this train control product line, feature *RegionalStation*, *Urban*, *Rural* and *ParkingTrack* are regarded as abstract features for their only use in creating hierarchies and facilitating better domain-specific understanding.

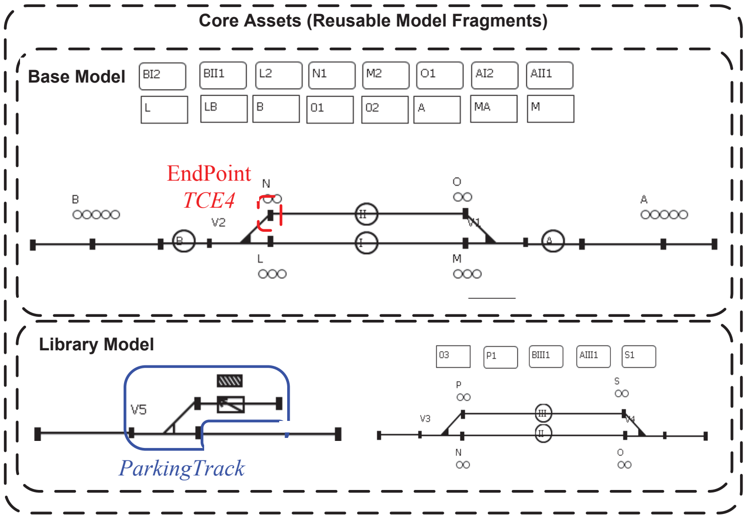


Fig.4. Core assets of the train control product line (including the base model and library model)

On the contrary, each concrete feature such as *AdditonalTrack*, *LeftParkingTrack*, *TopParkingTrack* and *RightParkingTrack* is supposed to be realized at the model (object) level by reusing/customizing the core assets of the product line. Fig.4 illustrates the core assets of this product line. In order to realize the feature *TopParkingTrack*, a possible model editing operation is to substitute the endpoint

TCE4 with the parking track (see Fig.4). Therefore the developer needs to explicitly specify this substitution in the definition of the realization for the feature *TopParkingTrack*.

When the development of a model-driven SPL is completed, the product line is ready for product configuration and product derivation. In order to derive a specific product model from the product line, the developer needs to choose a set of its required features (with associated realizations) from the product line model. This set of choices is called a "product configuration". During product derivation, realizations of a product configuration are executed through model-to-model transformations, to apply the feature realizations chosen in the product configuration process.

2.3 Evolving Model-Driven SPLs

Software product lines are often subject to changes to meet new requirements over time. Evolution in product lines can be identified into different categories depending on what the new requirements are. For a model-driven SPL, typical reasons for evolving a product line model include the following:

The core assets (reusable model fragments) are changed. Core assets are essential part of a product line and therefore can undergo frequent evolution (e.g., bug-fixes, refactoring, adding/deleting functionalities and etc. [119]).

Domain-level variability of a product line needs to be realized at the model (object) level. Furthermore, the specification of feature/variability realization should describe how to reuse/edit the core assets (e.g., in terms of a set of model editing operations). Therefore, variability realization also needs to be updated if it is affected by the changes in the core assets.

The metamodel of the base modeling language is changed. The core assets are reusable model fragments specified using the base language, therefore they may require changes in order to conform to the new metamodel. Subsequently the specification of variability realization may require changes as well.

New Product models need to be included in the product line. As a typical evolution scenario [26], augmenting a product line model to include new products has been mostly a manual process. It requires the developer to have a comprehensive understanding of the impact of every change to the existing product line model, so that both the new and the existing products can be derived from the augmented product line model.

2.4 Alloy

As elaborated in Section 6.2.2, one of our approaches in providing automatic assistance in defining variability realization, contributes to ensuring that the SPL will only yield intended products. As elaborated in Section 6.3.3, another approach of ours provides a semantic differencing technique for SPLs. In the feasibility evaluations of these two approaches, we used the Alloy language [67] and its tool support in the prototype implementation, for the formal analysis capability that Alloy provides.

Alloy [67] is a structural modeling language based on first-order logic for expressing structural constraints and behaviors. An Alloy module can consist of signatures, fields, facts, functions, predicates and assertions. Signatures denote sets of atoms. Fields belong to signatures and denote global relations between signatures. Relations are interpreted as tuples of atoms. Facts define global constraints. A predicate defines parameterized constraints, which will evaluate to true if all the contained constraints evaluate to true. A predicate can be regarded as an Alloy function whose return type is Boolean. An assertion is a claim that the contained constraints must hold.

The Alloy Analyzer [67] provides fully automated constraint solving for Alloy modules. All the modules are translated from first-order logic to propositional logic, which is analyzed by the Alloy Analyzer's embedded SAT solvers. The user needs to define a scope of the search space for the solver, namely a positive integer which limits the number of atoms for each signature that the solver should analyze.

Analysis in Alloy is based on the small scope hypothesis, which means that if there is a solution to a request, this solution will be in a scope of small size [10]. The Alloy Analyzer provides two types of analysis, one is to check if an assertion is valid, and the other is to find instances that satisfy a predicate, both in the user-defined scope.

2.5 EMF Compare

As elaborated in Section 6.2.1, we provide an automatic approach for synthesizing a product line from a set of existing products. The approach is built based on the CVL language and model comparison techniques. In order to evaluate the feasibility of the approach, we implemented a prototype tool, where EMF Compare[10] is used for its generic model differencing capability.

EMF Compare is a generic model differencing tool that can be applied to any two/three models specified in the same language which is defined in EMF. EMF is composed of the *MatchService* and the *DiffService*. During a model differencing process using EMF Compare, models are first interpreted into typed attribute graphs, and then fed into the match engine to identify matching model elements based on the overall score of four similarity metrics regarding the name, type, relations and content of the model element [33]. The match engine will output an *.emfmatch* model which lists all the matching model elements in the models under comparison. The *.emfmatch* model is further fed into the diff engine. The diff engine will go through the *.emfmatch* model and calculate the model difference based on it, outputting the differencing result in an *.emfdiff* model.

For example, for a two-way model comparison between a Left Hand Side (LHS) and a Right Hand Side (RHS) model, the *.emfdiff* model contains *unmatchedElements (left/right)* which represent the model elements that exist in the LHS/RHS model but not in the RHS/LHS model. The *.emfdiff* model also contains the *subDiffElements* of type *ReferenceOrderChange*, *UpdateReference* and *UpdateAttribute*, which represent

[10] http://www.eclipse.org/emf/compare/

the difference in reference order, reference and attribute value between two models respectively.

3 Research Topics

In the field of software engineering, there is a constant need for improved development tools and methods to support developing software systems of increased complexity. Model-Driven SPL development, combining MDE and SPLE, has emerged as a new paradigm for developing similar software systems.

Model-driven SPL development has inherited benefits from both MDE and SPLE paradigms. On one hand, model-driven SPL development raises the abstraction level from code to models; On the other hand, model-driven SPL development enables planned and managed reuse to improve the productivity of software development.

In order to maximize the benefits of model-driven SPL development, the developer should apply proper methods in all development phases. However, it is not always sufficient to "borrow" existing methods from MDE and SPLE paradigms. There is a need for methods and tools addressing problems that are specific to model-driven SPL development.

This thesis work has been funded by the MoSiS (Model-driven development of highly configurable embedded Software-intensive Systems) project. The goal of the MoSiS project includes: (1) Developing and standardizing a generic variability modeling language, and (2) Exploring whether the combination of MDE/DSM and SPLE can improve the existing software development process at industrial partners. The goal of this thesis work is based on the goal of the project and therefore focuses on contributing to the development of model-driven software product lines.

In particular, we identify the following research topics to improve the existing model-driven SPL development techniques, which are to be addressed in our thesis work:

3.1 Research Topic 1: A Generic Approach for Developing Executable Model-Driven SPLs (RT1)

A methodology is usually a guideline system for solving a problem. A methodology for model-driven SPL development should contain the study and description of a set of processes/guidelines for developing model-driven product lines. Many research works have proposed methods and tools for model-driven SPL development, such as variability/feature modeling techniques [109]. In contrast, only a few methodologies in this discipline have been proposed. Furthermore, we have identified several issues with the existing methodologies:

Few methodologies cover guidelines for the variability realization phase. The variability realization phase is an indispensable part of a complete product line development cycle. Without specifying how domain-level variability (features) should be realized at the model (object) level, no product models can be finally derived. However, most existing methodologies for model-driven SPL development do not include guidelines for specifying variability realization for product lines.

Few methodologies cover guidelines for the DSM phase. The developer cannot build a model-driven SPL without a base DSL and the core assets specified in this DSL. However, very few existing model-driven SPL methodologies include guidelines/processes for the DSM phase, which can be applied when a base DSL needs to be developed first.

Based on the issues stated above, we further detail this research topic by proposing the following research questions:

- **RT1.1:** How to define variability specification and realization in a generic way for model-driven SPLs in different domains?

- **RT1.2:** How to develop a base DSL suited for building model-driven SPLs?

3.2 Research Topic 2: Automatic Assistance in Model-Driven SPL Development (RT$_2$)

Automation is the use of machines, control systems and information technologies to optimize productivity in the production of goods and delivery of services. In particular, providing automatic assistance to model-driven SPL development can increase productivity in the production of software products. Various tools have been developed to provide automatic assistance in different phases of SPL development, however, the following issues are still not fully addressed:

Lack of automatic assistance in the variability identification phase. In order to define a product line, the developer needs to start with identifying the variability (and commonality) of all intended products of this product line. We learned from literature review that most existing techniques for variability identification are guidelines/methodologies without automatic tool support, such as how to survey domain experts/users (e.g., domain analysis [71]) and how to document/analyze variability from survey results (e.g., product map used in PuLSE [23]).

Lack of automatic assistance in the variability realization phase. The definition of variability realization directly affects the final product derivation. Therefore, it is crucial to ensure that variability realization will only yield intended products. From literature review, we see that, in the first place, most existing SPL development techniques do not cover the phase of variability realization, let alone providing automated assistance in specifying variability realization to ensure only intended product derivation.

Based on the issues stated above, we further detail this research topic by proposing the following research questions:

- **RT2.1:** How to improve the productivity of variability identification in model-driven SPL development by means of automatic assistance?

- **RT2.2:** How to ensure that variability realization will only yield intended products?

3.3 Research Topic 3: Evolving Model-Driven SPLs (RT3)

Software product lines are often subject to changes over time. From literature review, we have identified the following issues that are not fully addressed by the existing techniques for SPL evolution:

Lack of automatic tools for suggesting and performing SPL evolution from new requirements. Many existing automatic techniques in SPL evolution have their focus on managing and understanding product line evolutions, such as version control systems, program analysis and differencing tools. However, very few tools target on automatically suggesting and performing product line evolution based on new requirements from stakeholders.

Lack of automatic tools for SPL co-evolution. A model-driven SPL involves with several artifacts: the metamodel of the base DSL, the core assets which are reusable model fragments specified in the base DSL and the product line model. All these artifacts depend upon each other and all of them can subject to changes during product line evolution. Therefore, it can become necessary to co-evolve some of the other artifacts when one of them evolves.

For example, core assets is an essential part of a product line and therefore can undergo frequent evolution. Evolving the core assets of a product line may require co-evolving the definition of the product line model to ensure intended product derivation. Furthermore, when the metamodel of the base DSL evolves, the core assets may also require co-evolution in order to conform to the new metamodel. This may also subsequently bring the need to co-evolve the variability specification of the product line. However, very few existing techniques focus on providing automatic assistance in inducing and performing SPL co-evolution.

Lack of semantic differencing techniques to aid SPL evolution. It is common practice for the developer to compare the original and the evolved product line, in order to understand the impact of an SPL evolution. For this purpose, syntax-based differencing approaches have their limitations in situations when models of similar syntactical representation have very different semantics, which has been observed in feature models [6].

Small changes to a product line can result in big semantic difference in terms of derivable products. Therefore, it is crucial for the developer to gain an understanding on the semantic impact of product line evolution, in terms of which products have been added and removed in the evolved product line [3, 88, 99]. However, only a few approaches focus on semantic differencing for feature models.

Based on the issues stated above, we further detail this research topic by proposing the following research questions:

- **RT3.1:** How to improve the productivity in inducing SPL evolution steps from new requirements?

- **RT3.2:** How to improve the productivity of SPL co-evolution?
- **RT3.3:** How to assist the developer to gain a comprehensive understanding of the impact of an SPL evolution?

4 Research Method

This chapter gives a review of our technology research method and evaluation strategies. In addition, we also give a discussion on why we chose the method and evaluation strategies and how we have applied those in our thesis work.

4.1 The Technology Research Method

Solheim et al. [115] give the following definitions of technology and technology research:

"*Technology* is the knowledge of artifacts emphasizing their manufacturing".

"Technology research is research for the purpose of producing new and better artifacts".

We label our thesis as technology research, which is conducted by following the technology research method proposed by Solheim et al. [115].

The technology researcher focuses on seeking ideas for improving existing technologies and producing new and better artifacts. Solheim et al. [115] define technology research as a process iterating over the three steps:

Problem Analysis. In this step the researcher identifies and collects requirements for potential improvement to the existing technologies/artifacts, by means of literature review, surveying practitioners and etc. The requirements will serve as the goals for the manufacturing of a new and better artifact in the innovation step.

Innovation. In this step, the researcher starts to make an artifact which is supposed to satisfy the requirements collected from the problem analysis step [115].

Evaluation. In this step, the new artifact needs to be evaluated to see if the requirements for improvement have been satisfied, e.g. "*H*: The new artifact improves the efficiency of the current development process" [115]. However, since such hypotheses cannot be tested in straight-forward way, the researcher needs to formulate falsifiable predictions based on the requirements, e.g. "*P*: With the help of the new artifact, the programmer spends less time on the same task " [115]. Predictions are statements about what will happen if the hypothesis is true [115], e.g. if *H* is true, then also *P* will be true. Hence if investigations show that P is false, then the hypothesis H is rejected; if P is shown to be true, then H is confirmed.

However, in many cases, predicates cannot be falsified in a straight-forward way, such as in our example, we need to measure if less time is spent on the same task with the aid of the new artifact. Hence, the developer needs to carefully choose and apply the appropriate strategy for evaluation.

It is common for technology research to produce so-called functional prototype for evaluation [115]. If the prototype appears to be promising during the evaluation, it can be later elaborated/refactored to a product of commercial quality, which is typically performed by developers other than researchers.

4.2 How We have Applied the Research Method

The research method applied in this thesis work is based on the technology research method described in Section 4.1. The thesis work has been performed as an iterative process in which the artifacts and the requirements have been changed as we gained new inputs during the process.

Section 3 analyzes the purpose of this thesis work and further identifies the three research topics with the associated research questions. In the following, we describe in detail, in order to answer the research questions, how we follow the technology research method in identifying requirements for new artifacts, manufacturing and evaluation.

4.2.1 Problem Analysis

Literature review. This thesis work has been funded by the MoSiS project. The project goal is to explore and promote the combination of the DSM and SPLE paradigm. This overall goal has clarified the scope of our literature review.

We started the thesis work by conducting a state-of-the-art study on the subjects of DSM/metamodeling, variability modeling/SPLE and model-driven SPL development (see Section 5.1). During the literature review, we paid special attention to the areas in which our industrial partners had challenges to see whether those challenges can be addressed by existing artifacts (technologies) or not. If not, we further identified the requirements for new/better artifacts based on the need of our industrial partners and an in-depth analysis of the existing artifacts.

Surveys and Exploratory case studies. In the thesis work, we performed surveys and exploratory case studies for problem identification:

(1) *Surveys*. Survey research is used for identifying characteristics of a population of individuals [19]. It can be conducted by questionnaires, interviews or data logging techniques. A major challenge in survey research is the selection of a representative sample from a well-defined population, so that the results can be generalized from the sample to the entire target population [19]. It can be even more challenging to design survey questions in a way that can lead to useful and valid data. It can be difficult to ensure that all survey participants understand the questions in the same way. Moreover, participants may not answer the questions as they actually do if they do not introspect reliably on their common practices.

Survey research is less controlled and therefore lacks precision. Also if the sampling bias is not effectively controlled in a survey, the realism of the survey can be weakened. Moreover, if the participants for a survey are representative for the target population, the results of this survey can show high degree of generality.

During the span of the MoSiS project, we had frequent meetings with our industrial partners in different fields. During the meetings, we helped the industrial partners to identify problems in their daily software development and analyzed whether the problems can be addressed by means of model-driven SPL development.

(2) *Exploratory Case Studies*. Yin [138] defines case study as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident". Case studies are able to provide an in-depth understanding of why and how phenomena occur. In particular, case studies can be categorized into *exploratory case studies* and *confirmatory case studies*. Exploratory case studies are used for investigating phenomena to derive hypotheses and build theories, while confirmatory case studies are used to test hypotheses during evaluation.

In practice, we identified several candidate problems in the initial rounds of the meetings, and further investigated through exploratory case studies. We chose the method of exploratory case studies because it allowed us to gain an in-depth understanding on why and how the problems occur in real-life context at our industrial partners. During our exploratory case studies, we observed and interviewed at our partners on how the problems occur in their daily software development activities, as our means to collect data for further analysis.

For example, we had meetings with one of our industrial partners, Agresso, an ERP solution provider, to investigate whether their current development process can be improved from adopting the DSM paradigm. During meetings, we explained the concept of DSM to the developers from Agresso and discussed with them which part of their development can potentially be accelerated by applying DSM techniques. Developers from Agresso presented us with their problems on how to customize payroll reports for different customers efficiently. We performed case studies around this problem and concluded that such customization can be partially automated by applying DSM technologies, which results in our 2nd artifact ("APRiL") presented in Paper II [144] (Appendix II).

4.2.2 Innovation

In this phase, we developed new artifacts to address the challenges identified from problem analysis. The new artifacts aim to fulfill the requirements which existing technologies (artifacts) failed to satisfy. Our innovation efforts resulted in seven artifacts described by the papers in Appendices I-VII.

4.2.3 Evaluation

It is impossible in practice to choose an evaluation strategy that scores high on precision, realism and generality. According to Solheim et al. [115], the researcher needs to decide over the following factors when choosing evaluation strategies:

"Is the strategy feasible?" Time, cost and the availability of target participants are three important constraints when it comes to selecting an evaluation strategy. Therefore the researcher has to consider the feasibility of carrying out an evaluation study with respect to those three constraints.

"How to ensure that a measurement really measures the property it is supposed to measure?" It is critical to select an evaluation strategy which can be possible to isolate the property to be measured. In addition, the researcher also needs to account for all possible factors that might influence the result.

"What is needed to falsify the prediction?" It is not worthwhile to conduct an evaluation if it is not possible to falsify a result. Therefore the researcher needs to choose the evaluation strategy which is most likely to falsify the result, even though it would imply that the new artifact does not satisfy the need.

We have evaluated our artifacts through prototypes, confirmatory case-studies, examples, action research and formal analysis. The following gives a brief introduction on how we applied several evaluation strategies in our thesis work.

Prototypes and Confirmatory case studies. Our thesis work has been supported by several prototypes to evaluate the feasibility of concepts. The prototypes were further applied in confirmatory case studies to evaluate the validity of the new artifacts.

There are two critical steps in the design of case studies. Firstly, a precise *study proposition* needs to be formulated, which states the intention of the study and guides the selection of the cases and the collection of the data. Secondly, it is essential that the selected cases need to be the most relevant to the study proposition. Sometimes a single case is sufficient [138]: if the theory holds for a *critical case*, then it is likely to be true for many others; from an *extreme/unique case*, the researcher can gain insights on what happens in extreme situations; from a *typical case*, more insights into common situations can be gained. Nevertheless, a case study with multiple cases usually offer greater validity [138], either each case is expected to show the same result, or each case is expected to show contrasting results for predictable reasons.

Case studies are often applied where the context plays a role in the phenomena, or where the effects range widely or take long time to appear [138]. Case studies score high in realism because of its natural setting. However, because mostly qualitative data is collected during case study research which is susceptible to interpretation bias, case studies score low with respect to precision. When it comes to the concern of generality, case studies can score high if typical cases are used.

For the 2nd artifact ("APRiL"), DSL editors & code generators were developed for the APRiL language that we defined. Further we identified representative case-studies with the developers at Agresso and evaluated the prototype with the cases. In this way the developers at Agresso were able to try out our prototype in a natural work setting. The evaluation result was based on the observations and feedbacks collected from the case studies.

For the 1st artifact ("CVL") which is a generic and separate variability modeling language and the CVL methodology for SPL development, we developed an Eclipse plug-in as its prototype. This prototype has been distributed in both academia and industry, and has been validated against several examples in various domains. For example, we have applied the prototype on case studies at our industrial partners in the domain of train control, electrical drives, payroll reporting and etc.

The prototypes of our artifacts have been applied to various case studies (e.g., UML, TCL and APRiL) for evaluating the feasibility, performance and limitations of the approaches.

Action Research. In action research, the researchers attempt to solve a real-world problem while simultaneously studying the experience of solving the problem [135]. Different from just attempting to observe the world as it is, action researchers intervene in the studied situation with the purpose of also improving the situation. As a relatively new empirical method, it has been pioneered in the field of education and has been applied in software engineering on the studies of process/system improvement. For example, in order to evaluate the benefits of using UML in a professional software development environment, an action research can be conducted like this: if the researcher has professional programming competence, he/she can initiate a project to work with other programmers using UML and at the same time record the experience.

With him/herself also participating in the study, the researcher may gain more in-depth understanding of the studied situation. However, the generality of the results can be compromised if the researcher is not well-trained in collecting and analyzing data objectively.

Action researchers attempt to solve a real-world problem while simultaneously studying the experience of solving the problem. When applied in software engineering, action research is suitable for studying process/system improvement and introducing new development paradigms.

A prerequisite for conducting action research is that the researcher needs to have similar competence as other participants, so that the researcher will be able to participate in improving the situation while collecting experience at the same time. Since we have competence in both research and software development, we were able to apply action research method in our research. For example, in order to evaluate the benefits of adopting new paradigms (e.g., DSM and model-driven SPL development) in a professional software development environment, we worked with developers at our industrial partners using the prototypes of our new artifacts and at the same time recorded the experience. We carefully collected and analyzed the data to ensure the objectiveness and generality of our results. We were able to gain a in-depth understanding on if our new artifacts satisfy the needs in practice.

Formal analysis. Formal analysis is based on formal methods. Formal methods are mathematically based techniques for the specification, development and verification of software and hardware systems [38]. It is widely acknowledged that appropriate formal analysis can contribute to the reliability of a design. Formal analysis, as an evaluation strategy, scores high in generality and lacks realism and precision.

Formal analysis is based on the application of a variety of theoretical computer science fundamentals, such as logic calculi, formal languages, automata theory, program semantics and etc. For both our 4th artifact ("Automatic assistance in defining variability") and 7th artifact ("Semantic differencing for SPLs") (Appendix IV and VII), we utilized a formal language Alloy and the Alloy Analyzer (see Section 2.4) in the implementation of the prototypes. The Alloy Analyzer provides formal analysis based on first-order logic and embedded SAT solvers, which contributes to the reliability of our prototypes.

Potential weaknesses. When it comes to case studies, we used cases/examples provided by our industrial partners to ensure realism of our evaluation. However, the external validity of our case studies can still be potentially jeopardized by several factors.

For example, our cases/examples can be too narrow to cover all potential shortcomings, which may lead to biased conclusions. In order to address this problem, we tried to use representative cases/examples in various domains to ensure the generality of the results. However, this is limited to the availability of such cases for us. It was easier for us to access real cases from our industrial partners. Our major case studies in this thesis work have been performed with our industrial partners. In addition, our cases are typically small to medium-sized examples, and no industrial-sized examples have been used. This is also due to several factors: the availability of suitable industrial-sized examples, the time/cost limit to use such cases, and also the nature of the research – industrial-sized product lines as study candidates can be difficult to establish.

In order to ensure the validity of our case studies, on one hand, we focused on the representativeness when selecting cases/examples; on the other hand, we also applied our prototypes to cases/examples that have been widely used in academia. The validity of our research may be further strengthened by applying more quantitative/qualitative methods to collect statistical/descriptive data that can support our research claims.

Although it is possible to strengthen the validity of our evaluation, our main artifacts have been subject to evaluation as described in the corresponding research papers. In addition, the papers have also been evaluated by peer reviews where the soundness of the approaches has been considered. We have carefully considered the comments from the reviewers and improved our work accordingly.

5 State-of-the-Art

In this section we address the research works that are influential and related to what has been achieved in this thesis. In particular, we focus on Domain-Specific Modeling (DSM), variability modeling and model-driven SPL development, and evolving model-driven SPLs. Each area will be presented in the following manner: first we give a literature review of relevant research work, then elaborate on how we are motivated to improve the existing technologies in this area in regard to our research topics.

5.1 Variability Modeling and Software Product Line Engineering

In this section we discuss important work in the area of Software Product Line Engineering (SPLE) and how this is realized through variability modeling.

5.1.1 Developing DSLs suitable for Building SPLs

There are two strategies to specify a product line model:

The amalgamated approach, which is to extend the base language (e.g., UML or a DSL) with variability modeling language concepts.

The separate approach, which is to specify the variability of a product line in a separate variability model using a generic variability modeling language. The variability modeling language is defined beyond the base language of the product line.

Hence, when it comes to developing a DSL that is suitable for building product lines, the developer can either choose to have a more domain-specific DSL without variability modeling capability and leave that work to separate variability modeling approaches, or a DSL with both domain-specific and variability modeling concepts.

Including Variability into DSL Definition

Cengarle et al. [34] present a taxonomy of the variability mechanisms offered by modeling languages. As variability can be of presentation, syntactic and semantic nature, Cengarle et al. only talk about semantic variability. Furthermore, they propose a framework to explicitly document and manage variation points and variants specified in a variability modeling language. The framework facilitates systematic study of different kinds of variability and their dependencies. Moreover, it enables methodological customization of a language to a specific domain.

Morin et al. [91] propose to regard variability as an independent aspect to be woven into the DSL in order to introduce variability modeling capabilities. The approach is validated through the weaving of variability into two different metamodels: Ecore and SmartAdapter (an aspect model weaver [80]).

Ziadi et al. [148] extend the UML metamodel to include features for modeling variability. This work proposes extensions to model product line variability in UML class diagrams (the static aspect) and sequence diagrams (the behavioral aspect). Furthermore, this work also gives a formalization of product derivation using a UML model transformation.

Developing DSLs without Variability Modeling Capabilities

Paige et al. [97] provide a set of guidelines and recommendations to metamodellers and DSL designers in terms of factors that they should consider when constructing metamodels. One observation from this work is that a more expressive metamodel should not always be preferred over a less expressive one. The developer may need to make trade-offs between completeness and automation & usability of a DSL, depending on how the metamodel will be used in different tasks.

Karsai et al. [73] give a set of general guidelines to improve design and usability of DSLs. 26 guidelines has been proposed in this work, divided into five categories: *Language Purpose*, *Language Realization*, *Language Content*, *Concrete Syntax* and *Abstract Syntax*. Similar to Paige et al., they emphasize the importance of keeping the language definition simple and domain-specific, conveyed in the following three guidelines: "*Reflect only necessary domain concepts", "Keep it simple", "Avoid unnecessary generality"* and "*Limit the number of language elements*".

Kelly et al. [75] present guidelines for avoiding bad practices when developing DSLs. The guidelines emphasize the need for a comprehensive understanding of the domain in order to decide the correct level of abstraction and the correct scope for the DSL. They have analyzed numerous problem domains and metamodels. Based on this experience they give a set of general guidelines for DSL development. These guidelines emphasize the need for detailed knowledge of the domain. Their experience shows that it can be more challenging to extend an already existing language instead of creating a metamodel from scratch.

Wile [136] discusses experience from the development of two industry DSLs, including both success and failure factors. Based on this experience, twelve lessons have been given in what should be taken into consideration during DSL development. In particular, this work discusses the importance of using a notation that domain experts are already familiar with, and the need for develop a DSL closely with domain experts.

Motivation for Improvement

Based on the literature review, we see that there is potential to improve the state-of-the-art in relation to the research question *RT1.2 (How to develop a base DSL suited for building model-driven SPLs?)*. With most DSL development guidelines being rather general, there is a lack of guidelines for developing DSLs that are suitable for building product lines. However, we are still inspired by several general guidelines, especially when it comes to deciding whether the language should have variability modeling capabilities.

As pointed out by Paige et al. [97], Karsai et al. [73] and Kelly et al. [75], the developer should keep the language definition simple, domain-specific and just expressive enough to fulfill the current needs.

We also see that even with techniques to include variability into the DSL language definition, it may still have the following limitations in practice: (1) It may not be always feasible to change the definition of the base language and its tool support (editors, code generators and etc.). (2) The developer may prefer to keep the scope of the base language more domain-specific without offering variability modeling capabilities. (3) The developer needs repeat the work of extending the base language with variability modeling concepts when he/she starts building a product line with a new base DSL.

5.1.2 SPLE and Variability Modeling Approaches

Feature Modeling and Feature-Oriented Software Development

Software Product Line Engineering (SPLE) is an approach to produce the variability and commonality in a family of software systems sharing a common set of features [40]. A feature is regarded as "a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept" [43], e.g., additional functionalities for an existing base system [109].

Feature modeling has been widely used to capture and define the commonality and variability of systems in a SPL [45]. A Feature Model (FM) is represented by a Feature Diagram (FD) as a hierarchically organized set of features based on relationships among features [24].

Since the term "feature model" was first introduce by Kang et al. in the FODA case study [71], several extensions to the original FODA FM notation have been proposed in various studies [109]. We give a brief introduction on the basic and the extended feature modeling concepts in the following:

In the basic notation of feature modeling, features are hierarchically arranged based on the following relationships among them:

Mandatory. A child feature with a mandatory relationship to its parent feature is included in all the products which include its parent feature.

Optional. A child feature with an optional relationship to its parent feature can be included/excluded in all the products which include its parent feature.

OR. For a set of child features with an OR relationship to their parent feature, one or more of them can be included in the product which includes their parent feature.

XOR. For a set of child features with an XOR relationship to their parent feature, only one of them can be included in the product which includes their parent feature.

Implies. A feature implies another feature means that these two features must coexist in the same product.

Excludes. A feature excludes another feature means that these two features must not be chosen in the same product.

Over the years basic feature modeling concepts have been extended in the following aspects:

Feature cardinality. Czarnecki et al. [45] propose the concept of feature cardinality. A feature cardinality, denoted as *[n..m]* with *n* as lower bound and *m* as upper bound, indicates that minimum n and maximum m instances of this feature can be included in a product. Mandatory and optional features can be considered as special cases of features with cardinality *[1..1]* and *[0..1]* respectively.

Group cardinality. A feature group cardinality, denoted as denoted as <*n..m*> with n as lower bound and m as upper bound, indicates that minimum n and maximum m child features of this feature can be included in a product. Group cardinality can be regarded as the generalization of *OR* and *XOR* in the basic feature model notation.

Attributes. Attributes were introduced by Czarnecki et al. [43] into feature modeling as a way to represent a choice of a value from a large or infinite domain. It allows a feature to be associated with type, such as integer or string. A collection of attributes can be modeled as a number of sub features, where each is associated with the desired type.

Relationships. Several authors [61, 127] propose to extend feature models with different kinds of relationships such as *consists-of* or *is-generalization-of.*

Feature categories and annotations. FODA [71] distinguishes among context, representation and operational features. Griss et al. [61] propose functional, architectural and implementation feature categories. In FODA, additional information in terms of feature annotations includes descriptions, constraints, binding time and rationales. Feature annotations can also be priorities, stakeholders, default selections, open-or-closed-for-extensions attribute and exemplar systems [44].

Modularization. A feature diagram may contain one or more special leaf nodes as feature-model references, with each representing a separate feature diagram [45]. This mechanism allows breaking up large diagrams into smaller ones and reusing common parts in several places. Modularization is important for feature models which become too large to be considered in their entirety.

The definition of a feature has evolved over the years [14], from only representing abstract concepts of base domain, to concepts that need to be implemented in order to satisfy requirements [17, 20, 139]. Feature-Oriented Software Development (FOSD) favors systematic application of the feature concept in all phases of an SPL development life cycle [14]. FOSD makes the connection between specifying the product line in the problem space using feature modeling, and implementing the product line in the solution space [43].

The FOSD paradigm and the model-driven SPL development paradigm share several similarities. An FOSD lifecycle contains the following phases [14]:

Domain Analysis. This phase corresponds to the "variability identification" and "variability specification" phase in model-driven SPL development. At this phase, commonality and variability of the product line is identified and specified using feature modeling.

Domain Design and Specification. This phase corresponds to the phase of developing core assets of the product line (reusable model fragments) in model-driven SPL development. At this phase, essential structural and behavioral properties of the features are specified using a formal/informal specification and/or modeling language.

Domain Implementation. This phase corresponds to the "variability realization" phase in model-driven SPL development. In model-driven SPL development, feature realizations/implementations are defined as instructions for reusing the core assets (reusable model fragments). However, in the context of FOSD, mappings need to be established between features and source code. Several feature-oriented programming languages have been developed to address this requirement in FOSD, i.e. Jak [21], Feature C++ [15] and Xak [11] as feature-oriented extensions to Java, C++ and XML.

Product Configuration and Generation. This phase corresponds to the phase of configuring and deriving product models in model-driven SPL development. The difference is, at this phase in FOSD, source code of products instead of product models is generated. Furthermore, for FOSD, it is important to ensure the code correctness in the following aspect: 1) Syntactical-correct, namely that the generated code conforms to the syntax of the programming language. 2) Type-safe, namely that the generated code is well-typed according to the programming language's type system. 3) Behavioral-correct, namely that the generated system shows only intended behaviors.

Kastner et al. [74] presents Feature IDE, which is an open source framework for an Integrated Development Environment (IDE) for feature-oriented sofware product line development. Feature IDE provides support for the entire life cycle of a software product line, covering domain analysis, feature modeling, implementation and maintenance. Feature IDE supports a set of different feature/aspect-oriented programming tools including AHEAD [21], FeatureC++ [15], FeatureHouse [13] and CIDE [78], and thus provides support for many languages, including Java, C++, Haskell, C, C#, JavaCC and XML [74].

The work of FeatureIDE is close to OpenArchitectureWare (http://openarchitectureware.org), which provides an Eclipse-based open framework for developing model-driven software development solutions and DSLs [74].

Commercial solutions such as Pure::variants (http://www.pure-systems.com/) and Gears (http://www.biglever.com/) also provide support for software product line development, however, with more focus on domain analysis and less on variability implementation/realization.

Delta Modeling

Delta modeling is a language-independent approach for modeling system variability. In a delta-oriented SPL, a set of products can be represented as a *core model* and a set of *model deltas* [37, 63, 64, 107] . The core model represents a product for some valid feature configuration. The model deltas specify modifications required to apply to the core model in order to realize other features of the product line. The modifications include adding/removing/replacing model elements. Moreover, the model deltas contain *application conditions* which specify under which feature configuration the modifications should be carried out. The concept of application condition fulfills the need for establishing mappings between features and their realizations in SPL development. During *delta application*, a product model can be obtained by applying the model modifications contained in the chosen model deltas to the core model.

Orthogonal Variability Model (OVM).

Klaus Pohl et al. [99] propose Orthogonal Variability Model (OVM) approach to document variability across all software development artifacts/domain artifacts, including requirements, design, realization and test. With the OVM approach, the variability of a product line is documented explicitly in an OVM model, which is orthogonal to all domain artifacts. The following gives a brief introduction on the central concepts of the OVM approach:

Variation Subject. "A variation subject is a variable item of the real world or a variable property of such an item." and it answers the question of "why does vary? [99]" For example, "color" can be regarded as a variability subject which identifies a property of real-world items.

Variability Object. "A variability object is a particular instance of a variability subject. [99]" Examples of variability objects for the variability subject "color" can be red, black and grey.

Variation Point. "A variation point is a representation of a variability subject within domain artifacts enriched by contextual information. [99]" An example of a variation point can be the "color of a car" in the context of an automotive product line.

Variant. "A variant is a representation of a variability object within domain artefacts. [99]" "Red (cars)", "black (cars)" and "grey (cars)" can be considered as variants for the variation point "color of a car".

Variability Dependency. A variability dependency is the relation between variation points and variants. Each variation point must be associated with at least one variant. Each variant must be associated with at least one variation point. A variation point can have more than one variant. A variant can be associated with different variation points.

Optional Variability Dependency. The optional variability dependency indicates that the variant can be but does not have to be in a particular product if its associating variation point is resolved in this product.

Mandatory Variability Dependency. A mandatory variability dependency states that the variant must be part of a particular product if its associating variation point is resolved in this product.

Alternative Choice. "The alternative choice groups a set of variants that are related through an optional variability to the same variation point and defines the range for the amount of optional variants to be selected for this group." The *min* and *max* attribute of an alternative choice define the minimum and maximum number of variants that are allowed to be selected from this alternative choice group. For example, "red", "black" and "grey" is governed by an alternative choice with the *min* of value "1" and the *max* of value "1", namely that cars can only be in one of these three colors.

Variability Constraints. In an OVM model, the developer can define "excludes" and "requires" constraints between two variants, two variation points as well as one variant and one variation point.

Traceability between OVM model and domain artifacts. Domain artifacts (e.g., requirements, design models, code and tests) can be related to the variability defined in an OVM model by means of the following two types of dependency: (1) *Artifact Dependency between variant and development artifact.* A development artifact can but does not have to be associated with one or more variants. A variant must be related to at least one or more than one development artifact. (2) *VP Artifact Dependency between variation point and development artifact.* A development artifact can but does not have to be associated with one or more variation points. A variation point can but does not have to be related to one or several development artifacts.

Amalgamated Variability Modeling and SPLE using UML Techniques

The Unified Modeling Language (UML)[11] is a visual language for specifying, constructing, and documenting the artifacts of software-intensive systems. UML has become the de factor standard for modeling object-oriented systems.

Variability modeling techniques can be categorized into amalgamated and separate approaches. In the amalgamated approaches, the base language of the underlying domain, either a general purpose language like UML, or a DSL, is extended with variability modeling language concepts. In particular, UML can be extended with additional language concepts by introducing UML profiles for different purposes. The developer can specify variability in his/her UML models by annotating them with the terms provided by UML profiles for variability modeling purposes. Around the idea of applying UML for system design and variability modeling, a large body of UML-based SPLE approaches has been proposed [30, 31, 34, 55, 59-61, 82, 100, 147].

In particular, PLUS (Product-Line UML-Based Software Engineering) [59] is a SPL development method based on UML. The PLUS method puts together a UML profile to extend UML-based methods for designing single systems to handle software product lines, which cover the following processes:

SPL Requirements Modeling

- Use case modeling. With the UML profile provided by PLUS, the developer can annotate kernel, optional, alternative use cases and variation points for use cases in the UML use case model.
- Feature modeling. PLUS provides an approach for modeling and representing features in the UML notation, as well as a method to derive the feature model from the annotated use case model.

SPL Analysis Modeling

- Static modeling. In this step, a product line information model is specified to determine kernel, optional and alternative entity classes.
- Dynamic interaction modeling. In this step, interaction diagrams are specified to realize kernel, optional and alternative use cases.
- Dynamic state machine modeling. In the step, the developer specifies kernel, optional and alternative state machines.
- Feature/class dependency modeling. In this step, the developer determines the dependencies/mappings between features and kernel/optional/variant classes.

SPL Design Modeling

- Software architecture patterns. In this step, the developer determines the architectural structure and communication patterns for the product line.
- Component-based software design. In this step, the developer applies component-based software design methods to develop kernel/optional/variant components ports/interfaces and interconnections between components.

[11] http://www.uml.org/

Software Application Engineering

- In this step, first the required features for a product need to be chosen from the feature model, and then the application architecture of the product can be derived from the product line architecture and reusable components.

Bragança [31] presents MoDeLine, a methodological approach for model-driven development of software product lines. The MoDeLine approach is evolved from the 4SRS (4-Step Rule Set) method, which is a model transformational technique for obtaining system architectures from functional requirements specified as UML use cases for single systems. Based on 4SRS, the MoDeLine approach proposes the following adaptations for product line development:

(1) The approach extends the UML 2.0 metamodel and adopts activities to specify use case behaviors, so that each use case behavior can be specified using an activity diagram in MoDeLine. Furthermore, MoDeLine enables automated creation of use case realizations in the form of system architectural models (class/component diagrams), from functional requirements in the form of use cases and activity diagrams.

(2) The approach extends the UML-F profile [55, 100], a UML profile for frameworks, to include support for requirements and analysis models.

(3) The approach follows the notation for feature diagrams proposed by Van Deursen et al. [126]. In MoDeLine, the initial feature model can be automatically constructed from the use case model.

Motivation for Improvement

Based on the literature review, we see that there is lack of SPL development approaches which are based on separate variability modeling as well as allow the explicit definition of variability realization. Furthermore, in relation to our research question *RT1.1 (How to define both variability specification and realization in a generic way for model-driven SPLs in different domains?)*, we have been inspired by several existing works:

As pointed out by Bosch et al. [29], one issue in SPLE approaches is the lack of clear dependencies between features and the base model. Also we are enlightened by how choices and multiplicities are expressed in cardinality-based feature modeling.

When it comes to Orthogonal Variability Modeling (OVM) [99], we see the following limitations of the approach:

(1) In an OVM model, the developer can only define the dependencies between the variability of the product line and the development artifacts. The approach does not provide fine-grained model operations stating how development artifacts (models) should be modified in order to realize the selected variant.

(2) OVM only contains *excludes* and *requires* constraints. We see the need for supporting the specification of arbitrary variability constraints in a variability modeling language.

(3) OVM does not support specifying multiple instantiations of the same variant, which is covered by cardinality-based feature modeling.

When it comes to delta modeling [65], the modifications contained in the model deltas are limited to adding, removing, changing and replacing singular model elements. Delta modeling provides no flexibility to define the addition/removal/changing/replacement of an arbitrary set of model elements, which we see the need to provide in our work.

5.1.3 Variability Identification in SPL Development

Variability identification, as an integral part of SPL development, focuses on identifying the commonality and variability among all intended products of a product line before the actual building process. In this section we give a summary of the most related work in this area, as well as an analysis of how our thesis work was motivated by the state-of-the-art back in time.

Scoping Approaches in Product Line Planning
"Scoping can be defined as the process of deciding in which parts of an organization's products, features and domains' systematic reuse is economically useful" [70]. Scoping methods have been proposed in a number of SPL development methodologies:

Bayer et al. [23] present PuLSE (Product Line Software Engineering) as a product-centric and customizable methodology for the conception and deployment of software product lines within a large variety of enterprise contexts. As the scoping method for the PuLSE methodology, PuLSE-Eco proposes to use product maps, characteristic lists and benefit functions to assist decision-making at the scoping phase. Similar scoping approaches include: a commonality analysis technique proposed by Weiss [131], a commonality and variability analysis technique proposed by Chastek et al. [35], a product line potential analysis technique proposed by Fritsch et al. [56], a scoping method based on a decision-making framework [77], a collaborative approach for agile product line planning [95] and etc.

In the survey presented by John et al. [70], existing scoping approaches in the past years are identified and characterized with the goal of deriving open research questions. A few new research questions have been proposed, such as "*What is the influence of scoping on other software development phases?*", "*How is the connection between scoping and RE?*" and "*How is the connection between scoping and architecture?*"

Variability Extraction from Various Sources
As a new software development paradigm, SPL development has been increasingly adopted in practice. However, instead of building a product line from scratch, the organization often needs to build it based on existing products. In such scenarios, the product line should first include all existing products, and then possibly introduce new products. With the purpose of improving the productivity of variability identification in existing products, a large number of automatic/semi-automatic techniques have been proposed for extracting variability from existing artifacts (e.g., functional requirements, software models, source code, software architecture, product descriptions and formal descriptions).

Variability Extraction from Functional Requirements
Niu et al. [94] present a clustering framework for analyzing the functional requirements in an SPL. The framework provides automatic support during the variability extraction and clustering processes.

Weston et al. [134] present ARBORCRAFT, a framework for guided creation of feature models from requirements documents using natural language processing techniques. The ARBORCRAFT framework is developed based on an approach proposed in a previous feasibility study [8]. With ARBORCRAFT, the similarity of requirements will be measured and compared using the LSA (Latent Similarity Analysis) tool [116]. A feature tree will be created based on the results of the previous stage using a variant of HAC [36] technique for feature clustering. The EA-Miner [104] tool will be applied to detect variability in requirements to further refine only mandatory features into sub-features with dependencies if necessary. The resulting candidate feature model is subject to user input based on the user's domain knowledge and understanding of the requirement documents.

Variability Extraction from Software Architecture
Acher et al. [1] present a tool-supported approach to extract and manage the evolution of software variability from an architectural perspective in plugin-based systems. One of the focuses of the approach is automatic variability extraction from the software architecture of plugin-based systems.

The extraction process takes in the software architecture model, the plug-in dependencies and software architecture knowledge as input. The software architecture model consists of the set of elements needed to reason about the software system and the hierarchical relations among them. The plugin dependencies specify variation points and their logical dependencies supported by the architecture. The software architect knowledge can introduce accidental complexity and does not necessarily reflect how the software architecture is actually implemented.

On one hand, $fm_{Arch150}$ is extracted from a 150% architecture of the system, which consists of the composition of the architecture fragments of all the system plugins. The variability represented by $fm_{Arch150}$ is extracted by exploiting optional references in the architecture model.

On the other hand, fm_{Plug} is extracted from the plugin dependencies. The developer needs to specify a bidirectional mapping from $fm_{Arch150}$ and fm_{Plug} to show not only which plugin provides a given architecture feature, but also which architecture features are provided by a given plugin. In order to derive different fm_{Arch} which are feature models representing different set of configurations, firstly fm_{Plug} and $fm_{Arch150}$ are aggregated under a synthetic root in fm_{Full}, which also contains the mapping information between fm_{Plug} and $fm_{Arch150}$, then the subset of configurations of fm_{Full} are projected onto $fm_{Arch150}$ using a slicing operation.

Variability Extraction from Product Models
Lora-Michiels et al. [85] present an approach that integrates statistical techniques to identify commonality and variability in a collection of a non-predefined number of product models, which results in a automatically constructed product line model. The method consists of four steps:

(1) Preparation. In this step, a derivate matrix of feature occurrence in a collection of product models needs to be prepared. Features can be extracted from repositories by means of clustering techniques.

(2) Structural analysis. Step a) Firstly the collection of product models and the feature occurrence matrix are examined to identify structural patterns such as bundles, parents and sons, as well as a feature binary matrix. Step b) Once the feature binary matrix is built, the association rules data mining tool based on Apriori algorithm is executed to explore the association rules in the collection of product models. Step c) Identify mandatory relationships using association rules. The relationship is considered mandatory if at least one of the two association rules (high frequent feature and bidirectional rules) is fulfilled between a parent and child feature. Step d) Once mandatory relationships are identified, the remaining relationships between a parent and child feature may be classified as optional.

(3) Based on the feature binary matrix and parental relationships discovered from previous steps, a cross tabulation analysis and an independence test are performed to identify strong relationships, such as excludes and requires.

(4) Grouped cardinality analysis by means of identifying all possible feature sets for each bundle, and counting feature's occurrence in each product model.

Variability Extraction from Source Code/Legacy Systems

Ziadi et al. [146] propose an approach to automate feature identification from the source code of a set of product variants. With the approach, the input products are first abstracted as sets of construction primitives. Then feature candidates will be identified by means of an algorithm. In the end the set of candidates will be undergone manual edits to produce the final set of features of the product line.

Liu et al. [81] present a re-modularization approach for optimizing the synthesis of product families. The approach provides automatic support in: (1) Identifying shared files among products and extracting them into a common package. (2) Merging isomorphic class inheritance hierarchies into a single hierarchy.

Savage et al. [106] present *FLAT*$_3$, a tool suite for feature location. *FLAT*$_3$ allows the developer to locate features both textually and dynamically (monitor execution traces), as well as to visualize the dispersion of features or search results throughout a project.

Variability Extraction from Formal Descriptions of Product Lines

Gruler et al. [62] present an approach to model product lines in a formal manner. The approach allows computing the common parts of a product line (an entire PL-CCS program) in a well-defined way, and therefore facilitates matching components of the algebraic model with existing implementation artifacts.

Czarnecki et al. [47] present an approach for synthesizing feature diagrams from logical formulas, which produces a non-standard feature model with DAG structure. She et al. [114] improve the work by proposing a set of procedures for reverse engineering feature models based on a crucial heuristic for identifying parents, which is regarded as the major challenge for this task.

Andersen et al. [9] show that the problem of automatic synthesis of feature models from propositional constraints is NP-hard. In addition, this work also proposes a set of efficient techniques for improving the performance of synthesizing feature models from CNF and DNF formulas respectively.

Variability Extraction from Product Line Descriptions
Acher et al. [2] propose a semi-automatic process for extracting variability from a set of product descriptions organized in tabular forms. The approach provides automatic support in synthesizing a feature model by merging a set of products' descriptions.

Dumitru et al. [51] propose a system that models and recommends product features for a given domain. The approach mines product descriptions using a text mining and incremental clustering algorithm, in order to identify domain-specific features.

Variability Extraction using Formal Concept Analysis
Ryssel et al. [102] present a formal concept analysis-based approach for automatically constructing feature models from product variants that are given in the form of an incidence matrix.

The authors later propose another work based on OWL-based reasoning and formal concept analysis in [103]. This work allows the automatic verification of the feature mapping, as well as the automatic feature model synthesis for derived features which dependencies are not defined explicitly (e.g., Simulink variant objects).

Variability Extraction from Linux Kernel
Researchers have also worked on extracting variability from realistic examples, with the purpose of using realistic SPLs for benchmarking SPL approaches. She et al. [113] present the characteristics of the variability model of the Linux kernel, which can be extracted automatically from the Linux kernel configuration file. Dietrich et al. [50] present an approach for extracting variability from the Linux build system based on the make files.

Motivations for Improvement
Based on the literature review, we see that the existing technologies in variability identification have the following limitations, which correspond to our research question RT2.1: (1) Methodology/guidelines-based approaches lack automation support. (2) The applicability of some approaches is limited to specific languages such as UML. (3) It may not always be impractical in practice to build a product line from scratch. Therefore it is equally important to provide technologies for developing SPLs from existing products.

5.1.4 Variability Realization in SPL Development

During the variability realization phase of a SPL development, how the domain-level variability (features) of the product line should be realized/implemented during product derivation is specified. Variability realization can be carried out by various means (e.g., component-based software development [27], service-oriented implementation [128], feature-oriented code composition [13, 15, 22, 74, 78], composition/editing of domain models [31, 46, 59]).

Bosch et al., in [28], give a summary of different variability implementation/realization techniques from a high-level point of view. They suggest that the selection of a preferred variability realization technique should be driven by the binding time at which variants to be bound. Furthermore, the selection process should also take the following three factors into account, which are the mapping to the domain-level variability, the need for late-stage openness, and the expected system evolution.

Consistency Checking between Variability Realization and Variability Specification (Domain-Level Variability)

Since variability specification (domain-level variability) and variability realization reflect the problem space and solution space respectively, it is crucial to ensure that variability specification and realization are consistent with each other in the product line [46]. There are several research works which are intended for addressing this challenge.

Mussbacher et al. [92] propose an approach for detecting semantic interactions between aspect-oriented scenarios. In the process of semantic interaction detection, the approach applies critical pair analysis to semantic annotation in aspect models. In terms of tool support, the approach provides tool implementations for UML sequence diagrams and GRL goal models.

Ghanam et al. [57] present an approach to provide traceability links between the feature model and code artifacts to ensure the consistency between them. Executable acceptance tests are used as traceability links between features and code artifacts. A group of executable acceptance tests describe stories expected from a given feature of the system. Therefore the executable acceptance tests associated with a particular feature can be run through to see whether the current code artifacts (variability realization) have realized the domain-level feature.

Mohalik et al. [90] propose a formal semantics for SPLs using elementary set theory. This semantics makes it possible to give precise and unambiguous definitions to the traceability between variability specification and variability realization. Similar research efforts have been reported in [105], [39] and [41]. Satyananda et al. [105] propose an formal approach based on the PVS theorem prover for the verification of consistency between feature model and software architecture in the SPL; while Classen et al. [39] and Cordy et al. [41] focus on symbolic and non-boolean model checking of SPLs respectively.

Safe Composition

For compositional approaches, safe composition is the guarantee that all the programs, which can be composed based on an SPL's feature model, are type safe, i.e., without undefined references to classes, methods or fields [22, 83]. Safe composition is based on Czarnecki et al.'s observation that variability realization should reflect variability specification (domain-level variability) in an SPL [46]. Most research works in this area focus on the safe composition of source code [22, 79], while there is an increasing interest shown on the safe composition of software models [46, 84, 101] in the research community.

Thaker et al. [22] point out that, in a product line, low-level implementation of one feature can reference elements in the implementation of another feature. They present an approach for verifying if all the programs in a product line are type safe. Features are formalized into propositional formulas and the feature realizations (program segments) are analyzed to identify their dependencies between each other.

Czarnecki et al. [46] propose an approach for verifying feature-based model templates against well-formedness OCL constraints. Feature-based model template is an approach for model-driven SPL development. A feature-based template consists of a feature model and an annotated model that conforms to the metamodel of the base language. In the annotated model, variability of the product line is described in annotations (e.g., indicating that a specific model element can be optional in certain product models). With the feature-based model template approach, all possible product models can be derived by applying different changes to the annotated model, such as removing a model element which is annotated as "optional". The purpose of this approach is to verify that both the feature model and the annotated model are well-constrained so that all possible product models will conform to the metamodel and the constraints of the base language.

Kästner et al. [79] formally discuss a product-line-aware type system which is implemented with annotations on a common code base. Similar to Czarnecki et al.'s work [46], instead of checking all possible products of a product line in isolation, this approach checks the product line itself and ensure that all products from a well-typed product line are well-typed.

Motivation for Improvement
Based on the literature review, we see the following two challenges against deriving only products that are intended (see research question RT2.2):

(1) How to provide immediate feedback on the specification of variability realization at design time. Specifying variability realization is an error-prone process due to the complexity of the underlying domain. However, most existing SPLE tools do not provide the developer with immediate feedback on his/her specification changes at design time.

(2) How to ensure the consistency between the domain-level features/variability and their realizations. Domain-level constraints that govern the compatibility of features are often well captured in the product line model during feature specification, e.g., feature A implies B, indicating that these two features need to be included in the same product. However, the realizations of these two features may change the same base model element in different ways so that including the two features in the same product configuration will lead to conflicts during product derivation. We see the need for approaches detect such inconsistencies at design time.

5.2 Evolving Model-Driven SPLs

SPLs evolves over time to fulfill new requirements, e.g., to add/remove functionalities/products, to synchronize with the core assets/base modeling language of the product line that has been changed. In this section we give a brief summary of the most related work that has been reported in addressing challenges in SPL evolution, as well as an analysis on how our thesis work was motivated by those works.

5.2.1 Categorization of SPL Evolution

Svahnberg et al. [118] report on a case study of product line architecture evolution. Based on the case study, categorizations are proposed for the evolution of requirements, product line architecture and product line architecture components. In particular, to add products to a product line is categorized as one of the common requirements initiating an SPL evolution; while changed framework implementation is regarded as one of the common reasons for product line architecture component evolution.

Bosch et al. [27] report on a case study investigating the experience of component-based software development when product line architecture is presented. Based on the case study, the article discusses the difference between the academic and industrial view on software components, as well as the problems in using reusable components in product line architectures in industrial settings.

In particular, Bosch et al. [27] point out that, while reusable components are usually considered as black-boxes in research works, in real industrial cases they are often large pieces of software with a complex internal structure and no enforced encapsulation boundary, such as object-oriented frameworks. Also, while in research works components are often supposed to have narrow interface through a single point of access, in industry the component interface is often provided through entities (e.g., classes) which have no explicit difference compared to non-interface entities.

Elsner et al. [52] present an overview of approaches addressing "variability in time" which are time-related aspects in variability. The article identifies three types of "variability in time", which are: variability of linear change over time (maintenance/evolution), multiple versions at a point in time (configuration management), and binding over time (product derivation). The types are validated by using them to describe complex product line evolution scenarios where they exhibit expressive and discriminatory power.

Schmid et al. [108] present a taxonomy of requirements-based SPL evolution. The requirements-based SPL evolution is categorized into three levels: requirements level change, product level change and product line level change.

5.2.2 Augmenting Software Product Lines

Augmenting a product line has been so far mostly a manual process [4, 5, 16, 109, 111]. First, the product line developer checks if any of the new products are already included in the existing product line. If not, the developer is challenged to augment the existing product line in an optimal way so that: (1) The new products are incorporated, (2) The production of the existing products is not affected, (3) The changes to the product line are minimal. This requires the developer not only to perform an extensive comparison of the new and the existing products, but also to have a comprehensive understanding of the impact of any change during the augmentation process. Several research works can be applied to assist augmenting a product line with new products, which fall into the following categories:

Domain Analysis
Most of the domain analysis methodologies [23, 71, 72, 133] suggest a set of analytical means for the developer to manually identify the commonality and variability of the product line domain. In principle, those methods can be applied to analyze the commonality and variability between the new products and the existing products. However, since this requires extensive manual effort, the correctness of the augmented product line can be jeopardized if the domain analytical guidelines are not practiced diligently.

Merging Multiple Feature Models
Several works attempt to address the issue of merging multiple feature models (product lines) into one [4, 5, 16, 109] in the context of merger or cross-organizational cooperation. Those works mainly focus on defining the semantics of the merging operations.

Acher et al. [4] propose two operators for composing feature models. The *insert* operator allows inserting features from a crosscutting feature model to a base feature model. The *merge* operator enables the developer to merge features from two feature models which does not clearly crosscut with each other. When applying the insert/merge operators to compose large scale feature models, the developer is made aware of whether the current operation preserves the original semantics (set of product configurations) of the base feature model or not.

Apel et al. [16] present an algebra for features and feature composition. In this approach, the basic structure of a feature is modeled as a tree, called Feature Structure Tree (FST), which organizes the feature's structural elements such as classes, fields or methods hierarchically. Features are represented as FSTs and feature composition is expressed by tree superimposition and tree walks. Furthermore, a framework for feature composition has been implemented, which is fully independent of a concrete language.

To meet the need that a growing number of organizations produce and maintain multiple SPLs, Acher et al. [5] propose a compositional approach for managing multiple SPLs that involves automatically merging feature models across SPLs. The approach can be used not only to create feature models with certain product sets from multiple SPLs, but also to combine features from different SPLs to form products.

Segura et al. [111] propose an automated technique for merging feature models using graph transformations. In particular, the approach defines a set of visual rules to describe how to merge feature models and illustrates how those rules can be validated through tool support. Furthermore, the approach supports merging feature models with feature attributes and cross-tree constraints.

Automated Feature Model Construction
Research in this category focuses on how to suggest a feature model automatically from existing products, as summarized in Section 5.1.3. Other than automated feature model construction, those technologies can also be used to augment a product line with new products. First all the products of the existing product line need to be

generated, and then automated feature model construction techniques can be applied to these existing products together with the new products to synthesize a new product line model.

Motivation for Improvement

Based on the literature review, we see the following limitations in applying those merging techniques to augment a product line with new products: (1) It is not always practical to build a "delta" feature model from the new products first, and then merge it with the existing feature model. (2) The main stream merging techniques only support merger between two parent-compatible feature models. Two feature models are parent-incompatible if they contain features with identical names but differently named parent features. (3) Semantics of the merging operations only describe the set of properties that the merged feature model should have. Most of the works do not provide a mechanizable basis on how merging operations can be realized to automate the merging of feature models.

When it comes to apply automated feature model construction techniques to augment an SPL with new products, there are two drawbacks: (1) The amount of the existing products can be potentially high, thus it is not always practical to produce all the products from the existing product line for the synthesization of the augmented product line. (2) Manual work spent on the development of the existing product line may have to be discarded, e.g., the new product line may not preserve the overall hierarchy of the existing product line, which has been specified/inspected based on domain knowledge manually.

5.2.3 SPL Evolution Management

Domain Evolution in SPL

Adding new requirements to model-driven product lines often requires modifications to the product line’s core assets (reusable model fragments) and base language to reflect these new requirements. Since these modifications can involve much effort, automatic assistance is needed in deducing and performing such modifications.

Deng et al. [48] argue that when it comes to domain evolution in model-driven product line architectures, a layered and compositional architecture is needed to modularize system concerns and reduce the effort associated with domain evolution. Based on a case study, they illustrate that: (1) Structure-based model transformations can help maintain the stability of domain evolution by automatically transforming domain models. (2) Aspect-oriented model transformations can help to reduce human efforts by capturing model-based structural concerns.

Safe SPL Evolution

To safely evolve an SPL, it is important to assure behavior preservation of the original product line. Borba et al. [25] present a language-independent theory for product line refinement. The theory establishes refinement properties that justify stepwise and compositional SPL evolution. Similarly, Schulze et al. [110] extend the traditional definition of software refactoring to SPLs and propose the concept of variant-preserving refactoring of SPLs which are implemented using feature-oriented programming languages. According to this new definition, all variants of an SPL should remain valid after refactoring.

Neves et al. [93] discover and analyze concrete product line evolution scenarios. Based on the findings, a number of templates for safe product line evolution have been proposed, covering the evolution needs for splitting asset, refining asset, adding new optional feature, adding new mandatory feature and replacing feature expression.

Vierhauser et al. [130] report on their experience with a tool-supported approach for incremental and scalable inconsistency checking on variability models. They categorize inconsistencies within the problem space (i.e. feature models), solution space (i.e. UML models, domain models specified in base DSLs) and code space, as well as in between the spaces that may result from SPL evolution. The approach is extensive as new consistency constraints can be added. Furthermore, the approach is not limited to variability models but also applies to SPLs with concrete implementation, i.e. SPLs with underlying code base.

SPL Co-Evolution

A model-driven SPL deals with the following model artifacts which depend upon each other: the metamodel of the base DSL, the core assets which are reusable model fragments specified in the base DSL, and the product line model. These three types of model artifacts depend upon each other and together decide what product models can be derived from the product line. As the metamodel of the base DSL and core assets of the SPL are subject to evolution over time, it is crucial to ensure that product models can still be derived as intended. It may become necessary to co-evolve some other artifacts when one of them evolves. For example, evolving the core assets of a product line may require co-evolving the product line model to ensure intended product derivation. An effective model-driven SPL development cycle should provide support for SPL co-evolution [49].

Dhungana et al. [49] present an approach for supporting SPL evolution by organizing variability models of large scale product lines as a set of interrelated model fragments defining the variability of different aspects of the system. In addition to allow semi-automatic merging fragments into complete variability models, the approach also provides a metamodel change propagator which allows updating the existing variability models after changes made to the metamodel of the base DSL.

Seidl et al. [112] present the conceptual basis of a system for supporting the evolution of model-based SPLs, which maintains consistency between models and feature mapping from features to core assets. As part of their work, the authors introduce a classification of SPL evolutions based on the potential to harm the mapping of an SPL. Furthermore, with the purpose of co-evolving the feature mapping, several remapping operators have been proposed to rectify the negative side-effects of evolutions.

SPL Evolution Traceability

Passos et al. [98] have envisioned a feature-oriented project management and system development platform. As part of their vision, the platform supports traceability between features and the associated implementation artifacts. In addition, they believe that "organizing software evolution around features, supported by tracing, analyses and recommendations will address many of the challenges in understanding and managing change."

Mitschke et al. [89] propose a versioning model which enables traceability between features, artifacts (core assets) for feature implementation, and products in the context of software product line evolution. Serving as a basis for SPL evolution management, the approach provides traceability information that can ensure the consistency and maintainability of software product lines.

Jirapanthong et al. [68] present a rule-based approach to support automatic generation of traceability relations between feature-based object-oriented documents. In particular, the approach defines a traceability reference model with nine types of traceability relations (e.g., refinement and implement relation) for eight types of SPL artifacts (e.g., feature models, use cases and class diagrams). Furthermore, the eight types of SPL artifacts need to be specified in the document formats proposed by the FORM methodology [72], which in turn limits the applicability of this approach.

Anquetil et al. [12] present a model-driven traceability framework for software product lines. The approach identifies four orthogonal traceability dimensions in SPL development. The time dimension describes how an SPL artifact changes during evolution, which can be used to revert the changes caused by an evolution.

Motivation for Improvement

Based on the literature review, we see that the importance of providing automatic tool support for SPL co-evolution has been increasingly recognized. In particular, as pointed out by Seidl et al. [112] and Dhungana et al. [49], challenges may arise on how to update (co-evolve) a separate variability model (product line model) when its underlying core assets evolve over time. How can we ensure that the variability model is still valid? Updating the variability model according to the changed base model (core assets) can be a tedious task. However, there have been relatively few approaches which provide automatic co-evolution support in relation to variability model and core assets.

5.2.4 Semantic Differencing for Product Lines

Product lines evolve over time, and even small changes to a product line model can result in big semantic difference. It is vital for the developer to: (1) Identify the added and removed products in the evolved product line. (2) Check if all the products that the product line needs to offer to the customer are fully covered. (3) Document all the products that are supported by the product line, including those which are not offered to the customer yet [3, 88, 99].

Syntax-based differencing approaches have their limitations in situations when models of similar syntactical representation have very different semantics, which has been observed in feature models [24]. It becomes increasingly recognized that semantic differencing approaches can be more useful for certain purposes [6, 86, 87] such as understanding the impact of product line evolution.

Formal Semantics of Feature Models

Alves et al. [7] present a set of sound refactoring scenarios for feature models. As part of this work, the semantics of a feature model is formalized, which is defined as a set of product configurations that satisfy all the modeled constraints. Furthermore, the semantics is encoded using the Prototype Verification System (PVS) [96], which is a formal specification language.

Gheyi et al. [58] propose a theory for feature models in Alloy, which can be used to check a number of properties in the Alloy Analyzer [67]. As part of this work, the semantics of a feature model, which is specified in Alloy, is also defined as all the valid configurations that satisfy all the modeled constraints. Furthermore, the work also shows how to yield all valid configurations of a feature model in the Alloy Analyzer.

Similar work has been reported by Sun et al. [117], where the semantics of a feature model is specified using the first-order logic in Z [137]. The correctness of the semantics definition is validated using the Z/EVES theorem prover [124]. Furthermore, the semantics is also encoded in the Alloy analyzer for verifying the consistency of a given feature model. Our Alloy definition of the feature specification layer is mainly motivated by the definition of feature models in [58].

Semantic Differencing Techniques for Models

Fahrenberg et al. [53] present a formal approach for defining semantic difference between models. In the vision of this work, the difference between two models should be a model. In particular, a framework has been proposed for defining well-formed difference operators on model semantics as adjoints of model combinators, such as conjunction, disjunction and structural composition.

Archer et al. [6] present a set of differencing techniques for feature models, which render both syntactical and semantic mechanisms. The semantic difference between two feature models, represented as a semantic diff feature model, can be computed based on implication and exclusion graphs using SAT solvers.

Maoz et al. report their work on semantic differencing class diagram in [87] and activity diagram in [86]. They argue that the semantic difference should be a set of "diff witnesses", for instance, the diff witnesses of two class diagrams should be a set of object diagrams defined by the first class diagram but not the second.

Motivation for Improvement

Based on the literature review, we see the potential to improve the state-of-the-art in semantic differencing for product lines. In particular, existing semantic differencing techniques for feature models do not compare variability realizations of two feature models, which are considered crucial for final product derivation. Imagine the scenarios when the realization of a feature has evolved while the feature at the specification level remains the same, and vice versa. Semantic differencing for feature models fails to provide a complete picture of the impact of the evolution in this context.

Furthermore, we are also enlightened by Maoz et al.'s claim [87] [86] that the semantic difference should be a set of "diff witnesses". The concept of “diff witnesses”, if adapted to the context of product lines, can be well-suited for representing the semantic difference between two product lines in terms of derivable product configurations.

6 Contributions

The contribution of this thesis is manifested by seven artifacts developed by us (see Fig.5). Our artifacts are all concerned with model-driven SPL development, and mainly target on our three research topics. In this chapter we give an overview of the contributions, and we refer to Paper I - VII [122, 123, 141-145] (Appendix I - VII) for more detailed descriptions of the artifacts.

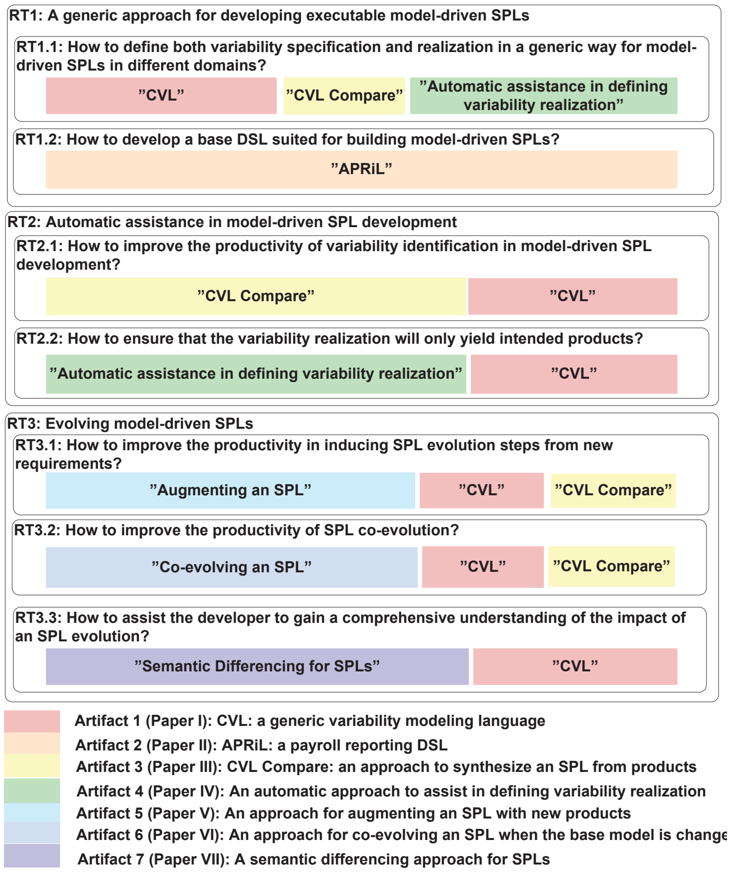

RT1: A generic approach for developing executable model-driven SPLs
RT1.1: How to define both variability specification and realization in a generic way for model-driven SPLs in different domains?
"CVL"
"CVL Compare"
"Automatic assistance in defining variability realization"
RT1.2: How to develop a base DSL suited for building model-driven SPLs?
"APRiL"
RT2: Automatic assistance in model-driven SPL development
RT2.1: How to improve the productivity of variability identification in model-driven SPL development?
"CVL Compare"
"CVL"
RT2.2: How to ensure that the variability realization will only yield intended products?
"Automatic assistance in defining variability realization"
"CVL"
RT3: Evolving model-driven SPLs
RT3.1: How to improve the productivity in inducing SPL evolution steps from new requirements?
"Augmenting an SPL"
"CVL"
"CVL Compare"
RT3.2: How to improve the productivity of SPL co-evolution?
"Co-evolving an SPL"
"CVL"
"CVL Compare"
RT3.3: How to assist the developer to gain a comprehensive understanding of the impact of an SPL evolution?
"Semantic Differencing for SPLs"
"CVL"
Artifact 1 (Paper I): CVL: a generic variability modeling language
Artifact 2 (Paper II): APRiL: a payroll reporting DSL
Artifact 3 (Paper III): CVL Compare: an approach to synthesize an SPL from products
Artifact 4 (Paper IV): An automatic approach to assist in defining variability realization
Artifact 5 (Paper V): An approach for augmenting an SPL with new products
Artifact 6 (Paper VI): An approach for co-evolving an SPL when the base model is change
Artifact 7 (Paper VII): A semantic differencing approach for SPLs

Fig.5. The overview of contributions

6.1 RT1: A Generic Approach for Developing Executable Model-Driven SPLs

In order to develop "a generic approach for developing executable model-driven SPLs (see RT1 in Fig.5)", we focus on the following aspects:

(1) "Generic" and "executable" require that, with this approach, the developer may not only define variability specification but also variability realization for SPLs in different domains, in a generic way. This requirement is addressed by our 1st, 3rd and

4th artifact (see Fig.5), which propose the Common Variability Language (CVL) and CVL-based methodology & automatic assistance for SPL development.

(2) As explained in RT1.2 in Fig.5, the developer needs guidelines on, when there is no base DSL yet, how to develop one that is suitable for building model-driven SPLs on top. We address this requirement by reporting our experience on developing both the base DSL and SPLs for the payroll reporting domain in the 2nd artifact ("APRiL").

6.1.1 RT1.1: How to define both variability specification and realization in a generic way for model-driven SPLs in different domains?

The Common Variability Language (CVL) is a separate and generic variability modeling language. CVL provides capabilities for defining both variability specification and realization in a generic way for SPLs in different domains, which contributes to RT1.1.

The initial results of the CVL language have been reported by Haugen et al. in [66]. We further developed the language and reported the new results in [54] and Paper I [123]. In Paper I we give a detailed introduction on the CVL language, and present a CVL-based methodology for SPL development through a realistic case study. In the following, we introduce the CVL language essentials in Section 6.1.1.1 and the CVL methodology in Section 6.1.1.2. We refer to Section 2.2 for explanations to the product line-related terms used in the following sections.

6.1.1.1 CVL Language Essentials

As a separate variability modeling language, CVL can be applied to models created in any DSL that is defined based on Meta Object Facility (MOF)[12]. When using CVL to develop product lines, the developer needs to deal with the following models (see Fig.6):

Base Model. A base model is a product model created in the base DSL. During product derivation, product models can be derived by applying feature realizations to the base model. The base model can be viewed as part of the product line's core assets (reusable model fragments).

CVL Model. A CVL model consists of the variability model and the resolution model:

- **Variability Model.** This variability model serves as the product line model, in which the SPL developer specifies the variability of the product line in this model. The term "variability model" is analogous to "feature model" in feature modeling.
- **Resolution Model.** A resolution model has one-sided relation to a variability model. Thus a variability model can have several resolution models. The developer can resolve the variability of the product line differently in several resolution models. Resolution models can be regarded as product configurations. The CVL generic transformation will take the base model

[12] http://www.omg.org/mof/

(and the library models if applicable), the variability model and the resolution models as input to generate resolved models, which can be regarded as the product derivation process. A "resolution model" is analogous to a "product configuration" in feature modeling.

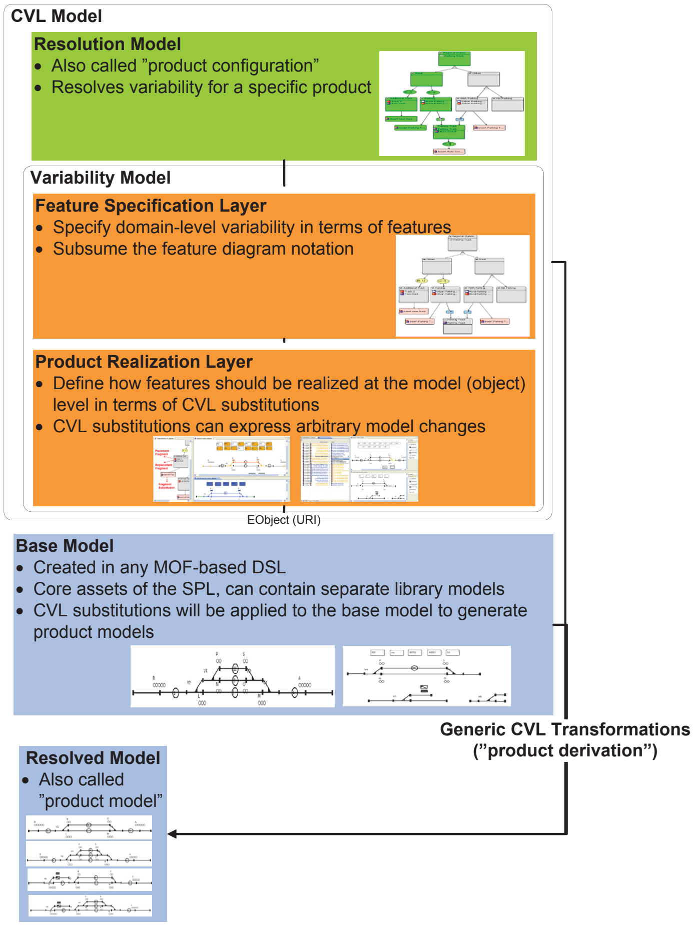

CVL Model
Resolution Model
• Also called "product configuration"
• Resolves variability for a specific product
Variability Model
Feature Specification Layer
• Specify domain-level variability in terms of features
• Subsume the feature diagram notation
Product Realization Layer
• Define how features should be realized at the model (object) level in terms of CVL substitutions
• CVL substitutions can express arbitrary model changes
EObject (URI)
Base Model
• Created in any MOF-based DSL
• Core assets of the SPL, can contain separate library models
• CVL substitutions will be applied to the base model to generate product models
Generic CVL Transformations ("product derivation")
Resolved Model
• Also called "product model"

Fig.6. Models in the CVL approach

In the variability model, the developer can specify the variability of the product line in two layers (see Fig.6):

Feature Specification Layer. The developer specifies domain-level features in this layer. CVL provides language constructs to specify features, relationships between features, multiplicity and choices over features. The specification in this layer can be regarded as a feature model.

In particular, the CVL language concept *CompositeVariability* can be used for modeling features in the feature specification layer of a CVL model. A *CompositeVariability* can also be further specialized into an *Iterator*, which can be used to model multiplicity and choices over features. *Iterator* has three properties: *upperLimit*, *lowerLimit* and *isUnique*. The first two properties specify the maximal and minimal number of features that are allowed to be chosen by this iterator. The property *isUnique* specifies if the same feature can be chosen more than once.

Product Realization Layer. This layer is not covered by the traditional feature modeling notation. In this layer, the developer defines how the features should be realized at the model (object) level in terms of CVL-specific model editing operations. These operations, called substitutions in CVL terms, can apply arbitrary changes of attribute value and structure to the base model to derive new product models during product derivation.

A CVL-specific substitution can be further specialized into the following substitutions:

ValueSubstitution, which changes the value of an attribute of a model element (*PlacementValue*) to another value (*ReplacementValue*) (see Fig.7).

ReferenceSubstitution, which redirects a reference from one model element (*PlacementObject*) to another one (*ReplacementObject*) (see Fig.7).

FragmentSubstitution, which substitutes an arbitrary set of model elements (*PlacementFragment*) with another set of model elements (*ReplacementFragment*) created in the same DSL. A *ReplacementFragment* can be defined either in the same base model or in separate library models.

Any arbitrary model fragment can be defined using *BoundaryElement(s)*. Boundary elements are used for recording all references to and from the model fragment. As illustrated in Fig.7, *ToP, FrP1* and *FrP2* define a *PlacementFragment*, whereas *ToR, FrR1* and *FrR2* define a *ReplacementFragment.* During a *FragmentSubstitution,* the boundary elements representing the *ReplacementFragment* need to be bound to the ones representing the *PlacementFragment*.

The developer needs to bind the boundary elements explicitly. Two boundary elements can only be bound if their recorded references are of the same type (the references point to the same type of model elements). For example, *ToR* is allowed to bind to *ToP* since both of their recorded references are of type *A*. Similar pairs include *FrR1* with *FrP1* and *FrR2* with *FrP2.* As illustrated in Fig.7, these three *Bindings* are the only legal choices; however, one boundary element can be eligible to bind to several as long as the typing rule is followed. The CVL tool can suggest default *Binding* candidates for each boundary element which are type-compatible. Nevertheless, with more than one eligible boundary element, it is up to the developer to decide on the final binding since only he/she knows how the resulting product model should look like.

An advanced CVL mechanism is to use configurable replacement fragment in substitutions. For example, the developer can define the value "*f*" of the attribute "name" of *f:F* (see Fig.7) as a *PlacementValue*. This variation point will be kept open after the *FragmentSubstitution* is executed. The developer can choose to assign a *ReplacementValue* in a *ValueSubstitution* to rename *f: F* at any point. A configurable replacement fragment can also have *PlacementObject(s)* and placement fragments inside.

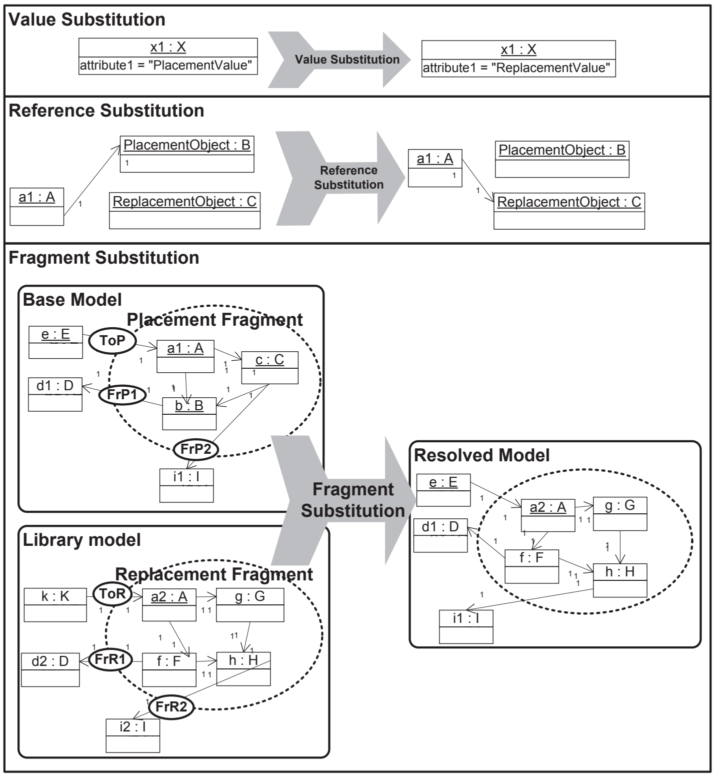


Fig.7. CVL substitutions

Based on the CVL language definition, we have developed a prototype to evaluate the feasibility of this artifact. The CVL metamodel is defined in EMF, from which a tree-view CVL editor is generated out-of-the-box. We have also developed CVL graphical editor (using GMF), a fragment substitution binding editor, a select-and-generate resolution model generator, a configuration validator and a generic CVL transformation (using MOFScript).

Furthermore, CVL provides a set of APIs for integrating different base DSL editors with the CVL editor. With a CVL-enabled base DSL editor, the developer can create placement/replacement fragments in the CVL editor automatically from the selection in the CVL-enabled base DSL editor, and boundary elements will be calculated automatically. Model elements involved in a substitution will be highlighted in the CVL-enabled base DSL editor when they are selected in the CVL editor.

6.1.1.2 CVL Methodology for SPL Development

In this section we walk through the process of creating the train control product line as described in Section 2.2 using the CVL methodology. The whole development process is illustrated in Fig.8. We refer to Paper I (Appendix I) for an industrial case study in developing a realistic train control product line, which all the products are real stations in use or under development in Norway.

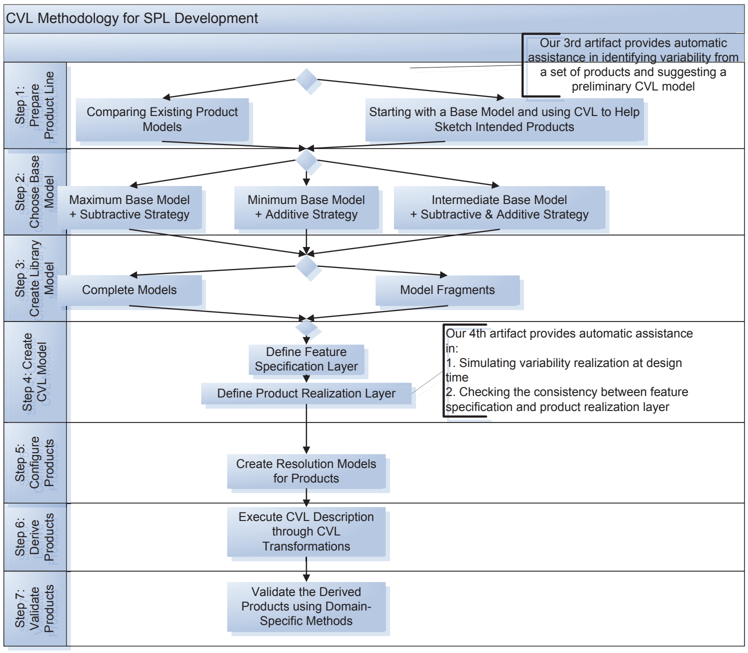

CVL Methodology for SPL Development
Our 3rd artifact provides automatic assistance in identifying variability from a set of products and suggesting a preliminary CVL model
Step 1: Prepare Product Line
Comparing Existing Product Models
Starting with a Base Model and using CVL to Help Sketch Intended Products
Step 2: Choose Base Model
Maximum Base Model + Subtractive Strategy
Minimum Base Model + Additive Strategy
Intermediate Base Model + Subtractive & Additive Strategy
Step 3: Create Library Model
Complete Models
Model Fragments
Step 4: Create CVL Model
Define Feature Specification Layer
Define Product Realization Layer
Our 4th artifact provides automatic assistance in:
1. Simulating variability realization at design time
2. Checking the consistency between feature specification and product realization layer
Step 5: Configure Products
Create Resolution Models for Products
Step 6: Derive Products
Execute CVL Description through CVL Transformations
Step 7: Validate Products
Validate the Derived Products using Domain-Specific Methods

Fig.8. CVL methodology for SPL Development

Step 1: Prepare Product Line

This step can be regarded as the variability identification phase in model-driven SPL development. The focus of this step is to capture the variability and commonality of all intended product models of the product line. In this step, there are different strategies for identifying variability depending on the context of the development:

When the product line needs to include existing products, it would be suitable to compare existing products to identify variability between them.

Our train control product line falls into this category. With the TCL language, the station drawings received from the Norwegian railway authorities can be re-created by train experts from ABB in terms of TCL models, from which source code for on-site signaling controllers can be generated. With the purpose of building a product line to include existing TCL (station) models, it is appropriate to start with comparing those station models which have been well-defined and validated.

The comparison results in the following understandings: stations can be either *Urban* or *Rural* ones depending on their location. Urban stations can have one *AdditionalTrack* compared to rural stations. Urban stations can also have a *LeftParkingTrack* and/or a *TopParkingTrack*. Rural stations can choose to have an optional *RightParkingTrack*.

Note that variability identification has been mostly a manual process. Our 3rd artifact ("CVL") aims to provide automatic assistance in the variability identification phase, which will be elaborated in Section 6.2.1.

When the product line is developed from scratch, the developer can start with choosing/specifying the base model (see Step 2). While defining the CVL model (see Step 4), the developer can then decide, relative to the base model, how the intended product models should vary from each other.

Very often a product line does not only need to include existing products, but also needs to introduce new products. We see that in this kind of scenarios, a combination of the two strategies mentioned above, namely that starting with comparing existing products, and further generalizing the product line to support more products, can be beneficial.

Step 2: Choose Base Model

The CVL model describes how the intended product models can vary from each other relative to the base model. During product derivation, feature realizations in terms of CVL substitutions will be applied to the base model to generate resolved/configured product models. There are different strategies for choosing/defining the base model for a CVL model (product line model):

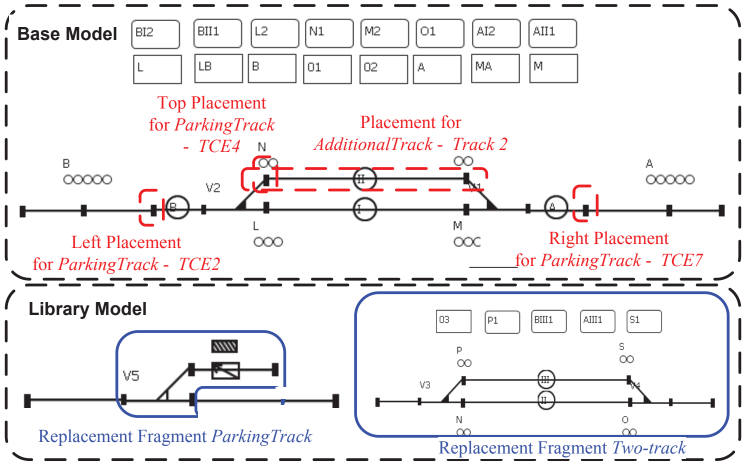
Subtractive strategy. In this category the base model includes sufficient model elements to cover all features of the product line. With a maximum base model, subtractive strategy will be applied while defining feature realizations. In other words, CVL substitutions will be solely removing elements from the base model while deriving product models. In this case, core assets for this product line only contain this base model.

Additive strategy. In this category the base model contains the minimum set of features, and other reusable model fragments are defined in separate library models (see Step 3). Then core assets for this product line will contain both the base model and library models. With a minimum base model, additive strategy will be applied while defining feature realizations (substitutions). Thus product models will be generated by adding features to the base model.

Combined strategy. In this category the base model is neither maximum nor minimum, but somewhere in between. This base model can be some model that is the most similar to the majority of all intended product models, or some model that is considered as a typical starting point for product development in industry. Then both subtractive and additive strategies will be applied while defining the feature realizations. CVL substitutions will be both removing and adding elements to the base model in order to generate product models.

How do we choose an appropriate strategy? In principle, no matter which strategy is applied in choosing/specifying the base model, as long as the CVL substitutions are defined correctly, the production of the SPL will be guaranteed. In addition, the product realization and feature specification layer of a CVL model is independent of each other. Therefore the production of the SPL will not be affected if naming of features does not quite reflect the purposes of their realizations.

However, different choices of the strategy may affect some non-functional factors, such as the readability, maintainability of the resulting CVL model, and how difficult it is to assign bindings of boundary elements while defining fragment substitutions.



Base Model
BI2
BII1
L2
N1
M2
O1
AI2
AII1
L
LB
B
01
02
A
MA
M
Top Placement for *ParkingTrack - TCE4*
Placement for *AdditionalTrack - Track 2*
B
N
V2
A
L
M
Left Placement for *ParkingTrack - TCE2*
Right Placement for *ParkingTrack - TCE7*
Library Model
V5
Replacement Fragment *ParkingTrack*
03
P1
BIII1
AIII1
S1
P
S
V3
N
O
Replacement Fragment *Two-track*

Fig.9. The base model and library model of the train control product line (with annotations)

As illustrated in Fig.9, we choose an intermediate base model for our train control product line. This station model has been created manually in the TCL graphical editor. We chose this station as the base model for the product line for two reasons, one is that it is the most similar one compared to all intended products; the other is that to the train experts at ABB, this station is a conventional starting point for station development in practice.

Step 3: Create Library Models

The CVL transformations derive a product model by making a copy of the base model and applying selected substitutions (feature realization) to it. When a model fragment in the base model is replaced by another model fragment in the library model, a copy is also made of the replacement fragment. Therefore the replacement fragment can either be taken from separate library models or the base model itself. When a minimum base model is used, separate library models may be needed in which some other reusable model fragments can be found.

With the TCL language, it is not possible to explicitly define model fragments such as *ParkingTrack* and *Two-track* (see Fig.9). However, while specifying the CVL model in the next step, the developer is able to define any arbitrary model fragment in the base and library model using boundary elements.

A library model can consist of either complete models where certain model fragments can be extracted, or only several model fragments. We recommend having library model fragments to be put in a more complete context (model). We notice that with some immediate context, when the replacement fragment is "cut off and taken" from its context (conceptually), the CVL fragment binding editor can be of more assistance in suggesting binding candidates that are type-compatible, and in turn might make it easier for the developer to decide on the bindings.

In our train control product line, we choose to have a separate library model with two model fragments *ParkingTrack* and *Two-track* (see Fig.9). This is due to the lack of existing complete models with these two required fragments.

Step 4: Create CVL Model

In this step the developer creates a CVL model (product line model) to specify the variability and commonality of the product line. In the feature specification layer, domain-level variability in terms of features is defined, while how the features should be realized in terms of substitutions is defined in the product realization layer.

As illustrated in Fig.10, a CVL model has similar notations as feature models with extensions and customizations. In the feature specification layer, *CompositeVariability* is used to model features such as *AdditionalTrack* and *ParkingTrack*, while *Iterator* is used to model choices over features, such as optionality, *XOR* and *OR*. In the product realization layer, fragment substitutions are used to realize the features. As illustrated in Fig.9 and Fig.10, feature *AdditionalTrack* can be realized by the fragment substitution which replaces *Track2* with *Two-track*; the feature *LeftParkingTrack* or *RightParkingTrack* or *TopParkingTrack* can be realized by replacing *TCE2* or *TCE7* or *TCE4* with *ParkingTrack.*

CVL provides a set of APIs for integrating with any base DSL editor. With a CVL-enabled base DSL editor, placement/replacement fragments can be created automatically in the CVL editor by selecting them in the CVL-enabled base DSL editor, boundary elements can be calculated automatically, and elements involved in fragment substitutions can be highlighted in different colors. As illustrated in Fig.11, the placement fragment *Track2* is highlighted in red (see the top right pane), while the replacement fragment *Two-track* is highlighted in blue (see the bottom pane). In addition, the elements that are referred from/to *Track2/Two-track* are highlighted in yellow and green respectively.

We see the need to ensure the correctness of the product realization layer in order to guarantee the derivation of all intended products in the later step. Our 4th artifact ("Automatic assistance in defining variability realization") provides automatic assistance in the creation of CVL models to ensure only intended product derivation, which will be elaborated in Section 6.2.2.

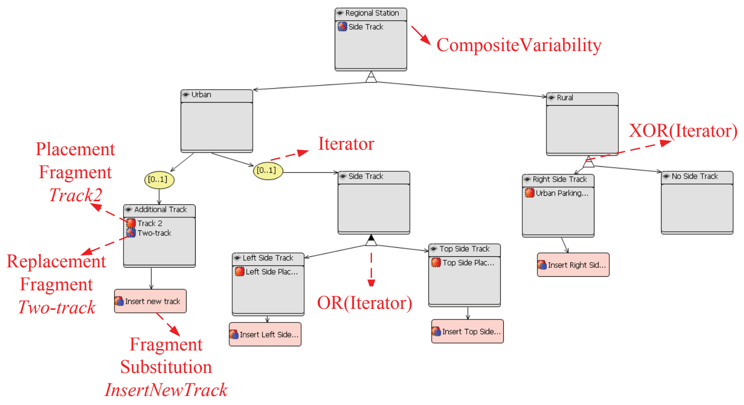


Fig.10. The CVL model for the train control product line (with annotations)[13]

Step 5: Configure Products

In this step the developer needs to create product configurations in terms of resolution models in CVL. Each resolution model specifies a set of features required for each product. A select-and-generated resolution model generator has been developed with CVL to automate the creation of resolution models. By selecting the required features directly from the CVL model, a resolution model representing a station with an additional track and a left parking track can be generated automatically. In addition, the CVL editor also supports highlighting any existing product configuration in green when requested.

Step 6: Derive Products

In this step the CVL model, the base and library model are input into the CVL transformations for product derivation. The following describes how a product model is derived during the transformations:

First a copy of the base model is made. Then the variability model is executed recursively by starting with executing the *ExecutablePrimitives* contained in the root *CompositeVariability*. If another *CompositeVariability* is contained in the current *CompositeVariability*, then *ExecutablePrimitives* contained in this *CompositeVariability* will be executed subsequently.

When the execution encounters an Iterator, it will stop, look up for the resolution of this choice in the resolution model, and continue again. When the execution encounters a fragment substitution, it will first remove the placement fragment from the copy of the base model, make a copy of the replacement fragment, and then place it into the "hole" in the copy of the base model following the definition of the boundary element bindings.

[13] The TCL terms "Side Track" and "Parking Track" are used interchangeably in this thesis.

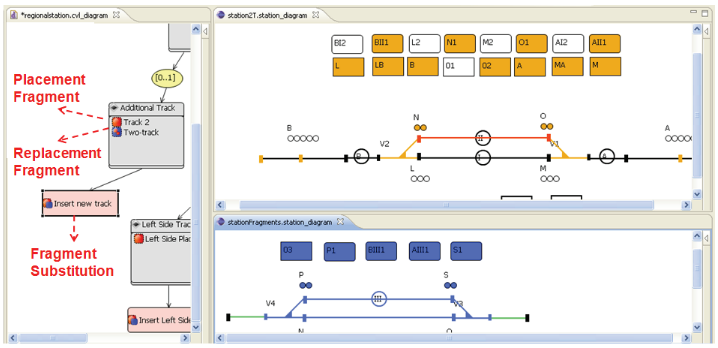


Fig.11. Fragment substitution for inserting an additional track (involved elements highlighted in the CVL-enabled TCL graphical editor)

Step 7: Validate Products

Since the generic CVL transformation is oblivious to the semantics of any base DSL, it is possible to generate semantically wrong products if the developer does not define feature realizations correctly. Therefore the derived products need to be validated using domain-specific methods, such as DSL editors, model validators and code generators.

When it comes to our train control product line, source code for on-site signaling controllers is automatically generated from all the derived station models, which is inspected against safety guidelines, and then executed on the simulator at ABB. If the source code goes through the simulation, then it confirms the validity of the derived stations.

6.1.2 RT1.2: How to Develop a base DSL Suited for Building Model-Driven SPLs?

In paper II (Appendix II), we have reported our experience on developing a base DSL and SPLs for the payroll reporting domain, in order to exemplify: (1) How to develop a base DSL that is suited for building model-driven SPLs (see RT1.2 in Fig.5). (2) How software development process can be simplified and improved by adopting the model-driven SPL paradigm.

We have developed the Agresso Payroll Reporting Language (APRiL) for the payroll reporting module of Agresso Business World © (ABW, an ERP system from Agresso, Norway). *Payment & Deduction* (P&D) is the basic payroll term in the ABW system, which is defined with codes by users. Typical P&Ds include *FixedSalary*, *OvertimePay*, *Bonus*, *Tax*. The value of each P&D can be calculated through ABW payroll transactions. The payroll of each employee can be calculated by summing up his/her P&D values.

For payroll reporting, users are often more interested in producing different views of payroll information instead of retrieving P&D values. For example, an employee may be interested in his/her actual or predicted salary for a certain period, while a manager may be interested in the human resource cost or average salary of a certain group of employees. However, it proved challenging for Agresso to provide a standard way to customize payroll reports in the ABW system, due to the following reasons:

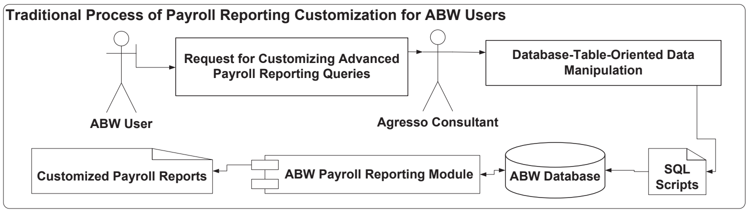

Traditional Process of Payroll Reporting Customization for ABW Users
Request for Customizing Advanced Payroll Reporting Queries
Database-Table-Oriented Data Manipulation
ABW User
Agresso Consultant
Customized Payroll Reports
ABW Payroll Reporting Module
ABW Database
SQL Scripts

Fig.12. The traditional process of payroll reporting customization for ABW users

(1) Payroll schemes can vary across roles in a company, and payroll schemes for the same role can also vary across companies. For example, payroll for a sales representative often includes commission while payroll for a secretary may not; an engineer may have bonus in one company but not in the other one.

(2) In order to customize advanced payroll reporting queries, the user needs to manipulate table columns and relationships in the ABW database directly using joins and filters. Since table relationships in the ABW database are not optimized for reporting purposes, it can be rather difficult for the user to create a complex reporting query in a short time. Normally in such situations, the user would seek help from the Agresso consultants who are supposed to have profound knowledge about manipulating the ABW database. As illustrated in Fig.12, the Agresso consultants need to manually produce SQL scripts for customized reporting queries, which can often be a time-consuming and error-prone process.

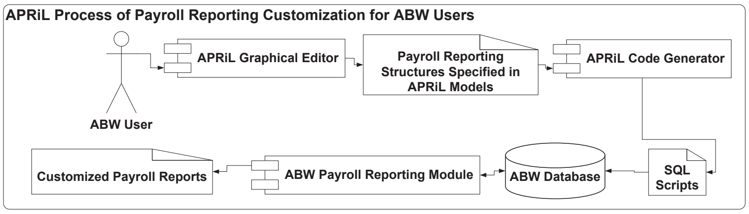

APRiL Process of Payroll Reporting Customization for ABW Users
APRiL Graphical Editor
Payroll Reporting Structures Specified in APRiL Models
APRiL Code Generator
ABW User
Customized Payroll Reports
ABW Payroll Reporting Module
ABW Database
SQL Scripts

Fig.13. The APRiL process of payroll reporting customization for ABW users

The APRiL language and tools have been developed using Eclipse technologies to address the challenges mentioned above. The metamodel of the language has been defined in Eclipse Modeling Framework (EMF), while the APRiL graphical editor has been developed using Graphical Modeling Framework (GMF) and the APRiL code generator has been developed using MOFScript. As illustrated in Fig.13, with

APRiL, the user is able to specify how a payroll report should be composed by P&Ds in the graphical editor, and the corresponding SQL script can be produced automatically from the code generator.

Note that the APRiL language can be used for expressing arbitrary payroll compositions. Moreover, the language is fully domain-specific and does not include any variability modeling concepts.

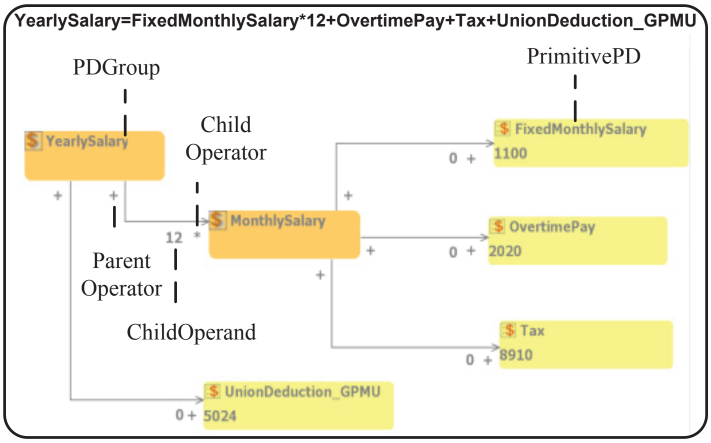


Fig.14. A payroll reporting structure specified in the APRiL graphical editor (with annotations)

In the APRiL language definition, *PaymentDeduction* (*P&D*) is the main concept in the payroll reporting domain. A *P&D* can be either a *PrimitivePD* or a *PDGroup*. Each *PrimitivePD* is defined independent of other *P&D*s and associated with a unique identifier in the ABW database. As shown in Fig.14, *FixedMonthlySalar*, *OvertimePay*, *Tax* and *UnionDeduction_GPMU* are all defined as *PrimitivePDs* with their unique IDs *1100*, *2020*, *8910* and *5024*.

PDGroup is a type of *P&D* that is defined dependent on other *P&Ds*. A *PDGroup* can consist of *PrimitivePDs* as well as other *PDGroups*, such as *YearlySalary* and *MonthlySalary* in the example. *PDGroups* are not defined in ABW database, but only used for aggregation purposes in payroll reporting. The hierarchies across *PrimitivePDs* and *PDGroups* make it possible to model payroll reporting structures in a more conceptual and intuitive manner.

With APRiL, relationships between *P&Ds* can be modeled explicitly using *PDGroupLink*. Moreover, the user can set the properties *ChildPDOperator*, *ChildPDOperand* and *ParentPDOperator* to each *PDGroupLink* to specify detailed composition of each *P&D*. Each *ChildPDOperator* and *ParentPDOperator* can be *PLUS*, *MINUS*, *MULTIPLY* and *DIVIDE*. As illustrated in Fig.14, *MonthlySalary* is the sum of *FixedMonthlySalary*, *OvertimePay* and *Tax* (*P&D* value is negative), while *YearlySalary* is calculated by multiplying *MonthlySalary* with *12* and adding it to *UnionDeduction_GPMU* (*P&D* value is negative).

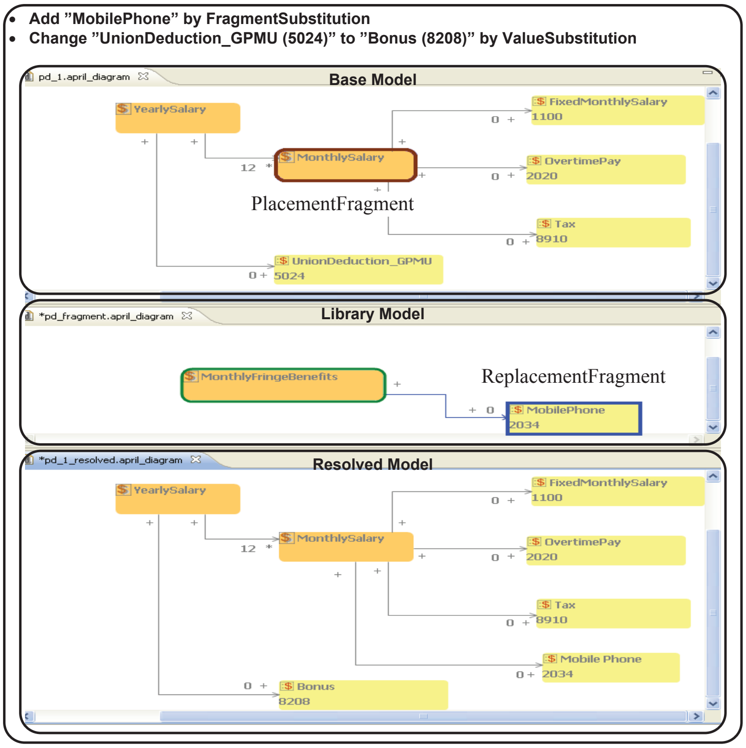


Fig.15. Base model, library model and resolved model for a payroll reporting example (with annotations)

In Paper II, we also illustrate how to add CVL-based variability handling to APRiL to further improve the productivity of the payroll reporting customization for ABW users. As illustrated in Fig.15 and Fig.16 , for a new payroll reporting structure similar to an existing one, instead of specifying the new structure from scratch, the developer can add separate variability handling (CVL-based) to the existing payroll reporting structure and generate the new one.

In order to evaluate the APRiL and APRiL+CVL approach, we have validated the generated script by comparing to the one that is manually developed by the Agresso consultants. Furthermore, we evaluated the use of APRiL on arbitrary payroll reporting structures provided by Agresso. Without knowing how to create advanced reporting queries in the ABW database using SQL, just like most users of the ABW system, we were able to create various payroll reporting structures using APRiL/APRiL+CVL approach in a short time. The generated SQL scripts were first inspected by the Agresso consultants, and then executed to see if those queries return the expected results.

We have learned the following lessons from this experience: (1) DSM and model-driven SPL paradigms have big potential in improving the productivity of software development by shifting certain responsibilities from the developer to the domain expert. (2) In order to adopt model-driven SPL development with separate variability handling, it would be ideal that the base DSL is made to be fully domain-specific without any variability modeling concept.

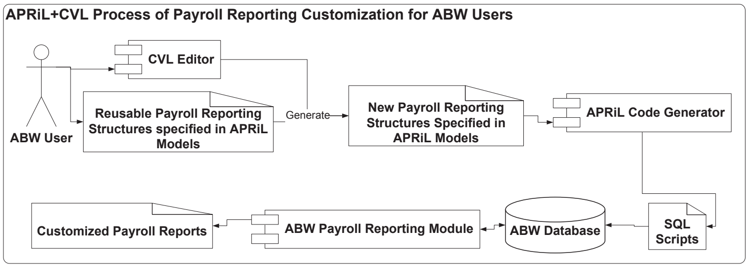

APRiL+CVL Process of Payroll Reporting Customization for ABW Users
ABW User
CVL Editor
Reusable Payroll Reporting Structures specified in APRiL Models
Generate
New Payroll Reporting Structures Specified in APRiL Models
APRiL Code Generator
SQL Scripts
ABW Database
ABW Payroll Reporting Module
Customized Payroll Reports

Fig.16. The APRiL+CVL process of payroll reporting customization for ABW users

6.2 RT2: Automatic Assistance in Model-Driven SPL Development

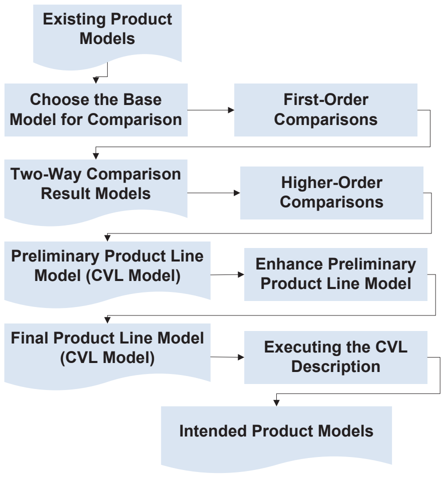
There is a lack of methods providing automatic assistance at the variability identification and variability realization phase of model-driven SPL development. In Section 6.2.1 and Section 6.2.2 we summarize how our artifacts contribute to RT2.1 and RT2.2.

6.2.1 RT2.1: How to improve the productivity of variability identification in model-driven SPL development by means of automatic assistance?

The life cycle of SPL development usually starts with identifying the variability (and commonality) of a product line's all intended products. We learned from literature review that most existing techniques for variability identification are guidelines/methodologies without automatic tool support.

We also noticed that in practice not all product lines are developed from scratch. Often the developer needs to include existing products in a product line and further introduces new products on top of that. Based on these observations, we identified the requirement for synthesizing a product line from a set of existing products automatically. We presented our initial ideas at a PhD symposium session (ICSE 2009 [140]), where we received positive feedback from experts in the field. Based on the comments, we have realized the idea and developed our 3rd artifact – CVL Compare.

CVL Compare [142] is an approach for synthesizing a product line model from a set of existing product models. It has been developed based on the CVL technology and a generic model differencing tool – EMF Compare. The CVL Compare tool enables the automation of variability identification phase. It is able to identify the variability and commonality of the product line and suggest a preliminary product line model automatically. The preliminary product line model can serve as a baseline for further manual enhancement.



Existing Product Models
Choose the Base Model for Comparison
First-Order Comparisons
Two-Way Comparison Result Models
Higher-Order Comparisons
Preliminary Product Line Model (CVL Model)
Enhance Preliminary Product Line Model
Final Product Line Model (CVL Model)
Executing the CVL Description
Intended Product Models

Fig.17. The process of the CVL Compare approach

Fig.17 illustrates the process of the CVL Compare approach, which will be walked through in the following using an example in the train control domain. The purpose of the example is to build a train control product line based on four existing TCL station models as illustrated in Fig.18. Note that the following is a simplified summary of the whole process. We refer to paper III (Appendix III) for a detailed description of the approach.

Step 1: Choose the Base Model for Comparison

In this step the developer needs to choose the base model for the CVL Compare process. The base model can either be chosen from the given set of existing product models, or a product model different from all of them. The chosen product model will also serve as the base model for the preliminary product line model (CVL model). Therefore the developer can follow the same strategies for choosing the base model as described in Section 6.1.1.2. When it comes to our example, we choose *S1* (see Fig.18) as the base model.

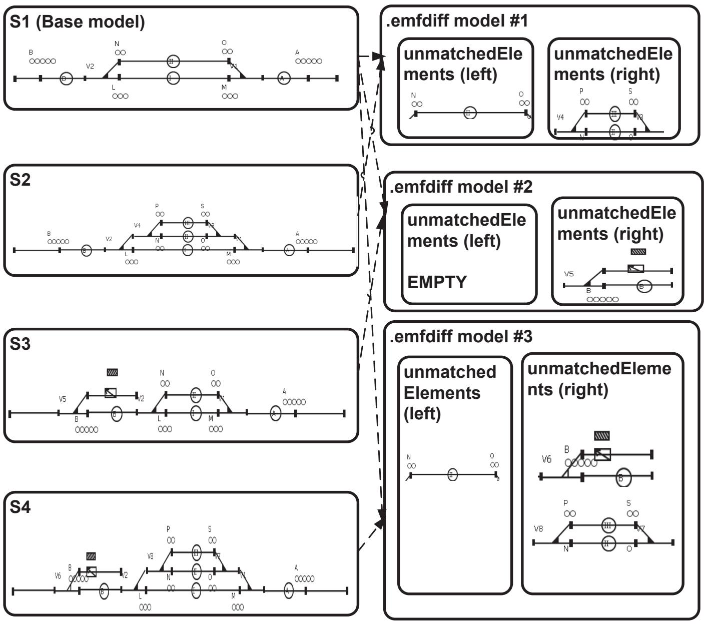


Fig.18. First-order comparison between four TCL models

Step 2: First-Order Comparisons

In this step the CVL Compare tool utilizes EMF Compare (see Section 2.5) to perform a set of two-way comparisons between the base model and other existing product models. As illustrated in Fig.18, *S2*, *S3*, *S4* are compared with *S1* respectively, which result in three difference models in the form of *.emfdiff #1*, *.emfdiff#2* and *.emfdiff#3*.

In an *.emfdiff* model, *unmatchedElements (left/right)* are the model elements that only exist in the left-hand-side (LHS)/right-hand-side (RHS) model but not in the RHS/LHS model. As illustrated in Fig.18, the second track exists in *S1* but not in *S2*, and the two-track structure exists in *S2* but not in *S1*. This result is used by the CVL Compare tool for the following CVL-specific interpretation:

(1) The RHS model can be obtained from the LHS model by replacing the *unmatchedElements (left)* with the *unmatchedElements (right)*. Therefore the second track and the two-track structure can be regarded as a placement/replacement fragment respectively.

(2) An *.emfdiff* model consists of *subDiffElements* of type *ReferenceOrderChange* and *UpdateReference*, which can be regarded as the reference changes due to the bindings of boundary elements in a fragment substitution.

(3) An *.emfdiff* model consists of *subDiffElements* of type *UpdateAttribute*, which can be regarded as value substitutions in CVL.

Step 3: Higher-Order Comparisons

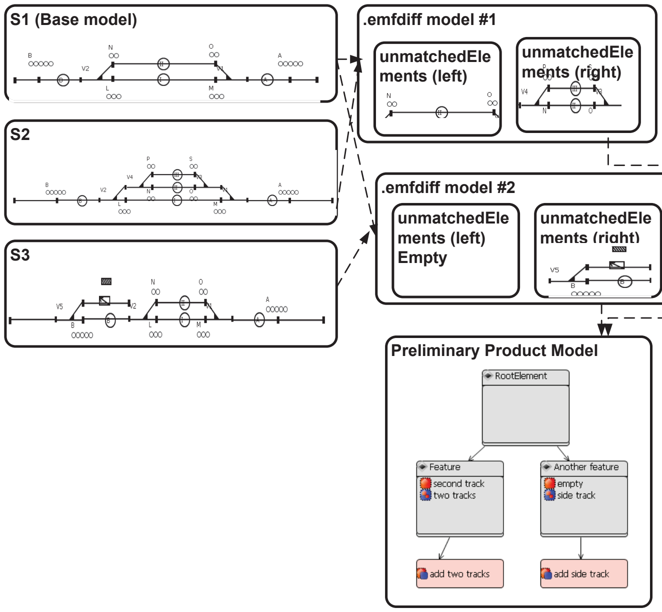


Fig.19. Higher-order comparison between for Station *S2* and *S3*

In this step all *.emfdiff* models resulted from the first-order comparisons are compared with each other, which are called higher-order comparisons. A preliminary product line model (CVL model) will be generated based on the result of higher-order comparisons at the end of this step.

First of all, an empty CVL model is created with its base model as the one chosen in Step 1. Then this CVL model will be updated based on a set of rules as the higher-order comparisons continue. We refer to Paper III (Appendix III) for a detailed description of the rules.

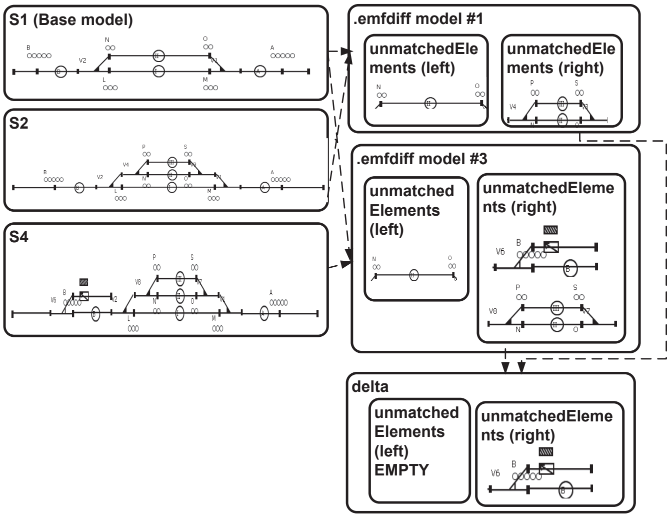

S1 (Base model)
S2
S4
.emfdiff model #1
unmatchedElements (left)
unmatchedElements (right)
.emfdiff model #3
unmatched Elements (left)
unmatchedElements (right)
delta
unmatched Elements (left) EMPTY
unmatchedElements (right)

Fig.20. Higher-order comparison between for Station *S2* and *S4*

Fig.19 illustrates how the CVL Compare tool performs the comparison between *.emfdiff#1* and *.emfdiff#2*. The CVL Compare tool first compares the *unmatchedElements (left)* in both models. It is found out that the *unmatchedElements (left)* in the *.emfdiff model #1* is a structure of second-track, and there are no *unmatchedElements (left)* in the *.emfdiff model #2*. Since this implies a potential distinctive variation point, two features are created on the same hierarchy of the CVL model (see Fig.22). At the same time, in the product realization layer, two placement fragments are created from the base model elements which are referred by the *unmatchedElements (left)* in both models. Similarly, two corresponding replacement fragments, which contain a two-track structure and a parking track respectively, are created from the *unmatchedElements (right)* in both models. In addition, two corresponding fragment substitutions *InsertNewTrack* and *InsertParkingTrack* are created[14] and the bindings of boundary elements are decided automatically.

As illustrated in Fig.20, when *.emfdiff#1* and *.emfdiff#3* are compared, these two models are found out to contain the same variation point, represented by the second track. This variation point is already identified in the comparison of *.emfdiff#1* and *.emfdiff#2*.

[14] The fragment substitutions are created with automatically-generated names by the CVL Compare tool. We refer to them as *InsertNewTrack* and *InsertParkingTrack* for presentation purposes.

The *unmatchedElements (left/right)* in both models are compared which results in a delta. The current CVL model is searched through to see if the delta suggests any existing placement/replacement fragment.

The comparisons also show that the parking track and the two-track structure can coexist in the same station model, suggesting a possible *OR* over the two features that they represent.

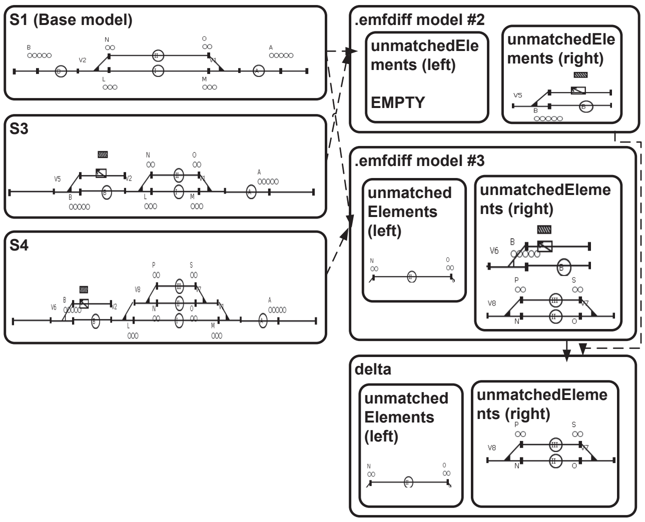

S1 (Base model)
S3
S4
.emfdiff model #2
unmatchedElements (left)
EMPTY
unmatchedElements (right)
.emfdiff model #3
unmatched Elements (left)
unmatchedElements (right)
delta
unmatched Elements (left)
unmatchedElements (right)

Fig.21. Higher-order comparison between for Station *S3* and *S4*

As illustrated in Fig.21., *.emfdiff#2* and *.emfdiff#3* are compared in a similar way. After all the higher-order comparisons, the resulting CVL model serves as the preliminary product line model as illustrated in Fig.22. This CVL model only defines the variability based on the given four station models, which is: a station can either only have two tracks, or with an additional and/or parking track.

Step 4: Enhance the Preliminary Product Line Model

In this step the preliminary product line model is enhanced manually by the developer. Typical enhancements include renaming features, restructuring and adding new features to introduce new products.

In paper III we also discussed the challenge of identifying compound variability in our approach. With CVL Compare, the difference between two product models is represented as a fragment substitution with necessary value substitutions. The placement/replacement fragment contains all the elements that exist in one model but not in the other one and vice versa. Nevertheless, the placement fragment may suggest more than one variation point that we cannot identify by only comparing two models without having any additional information.

One possible solution is to incorporate the semantics of the base domain in the CVL Compare tool so that some multiple variation points might be identified based on both comparisons and domain knowledge.

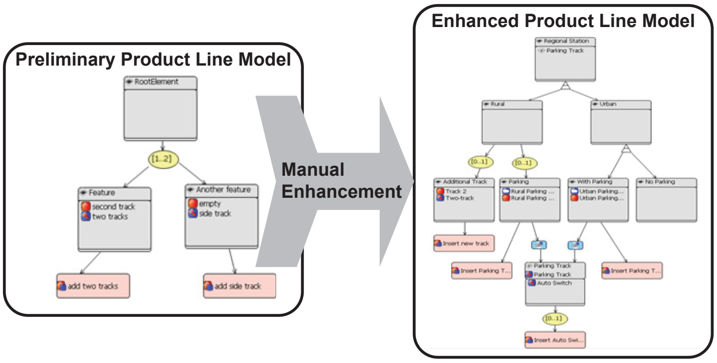


Fig.22. The preliminary product line model suggested by CVL Compare and the manually enhanced product line model

6.2.2 RT2.2: How to ensure that variability realization will only yield intended products?

The definition of variability realization directly affects product derivation. Therefore, it is crucial to ensure that variability realization is properly defined so that it only yields intended products during product derivation (RT2.2).

Our 4th artifact (Appendix IV) contains two methods with automatic tool support for the variability realization phase in SPL development to ensure intended product derivation. We summarize how the two methods contribute to RT2.2 in Section 6.2.2.1 and 6.2.2.2.

6.2.2.1 Providing Immediate Feedback on the Definition of Variability Realization at Design Time

In a CVL-based product line, variability realization is defined in the product realization layer in terms of CVL substitutions. For a CVL fragment substitution, default bindings of boundary elements which are type-compatible can be automatically suggested by the CVL tool. Based on that, the developer still needs to decide on the final bindings explicitly, since only he/she knows how the resulting model should look like after applying the substitution. However, the developer does not necessarily have an accurate mental picture at design time on how the current definition of the substitution (bindings) will change the base model at the model (object) level. We saw that defining variability realization can be an error-prone process and there was a lack of immediate feedback for the definition of variability realization at design time.

In Paper IV (the 4th artifact), we addressed this challenge by extending the CVL editor with a simulator, which can simulate the execution of a single CVL substitution at design time. The result of the simulation is visualized as the abstract syntax of the resulting model excerpt (with only related model elements).

The simulator is developed based on CVL and Zest[15]. It is generic so that it can be applied to models created in any MOF-based DSL. However, the generality does come with a cost. With no knowledge about the concrete syntax of the base DSL beforehand, it is impossible for the simulator to represent the resulting model excerpt in the domain-specific form. Therefore, we chose to represent the preview of the simulation result in the abstract syntax model with additional domain-specific information, such as name/type of model elements/references/attributes.

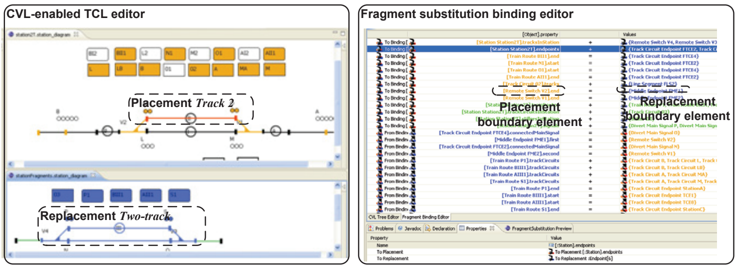

CVL-enabled TCL editor
Placement Track 2
Replacement Two-track
Fragment substitution binding editor
Placement boundary element
Replacement boundary element

Fig.23. Fragment substitution "Insert new track" shown in the CVL-enabled TCL editor and fragment substitution binding editor

In the following we illustrate how to define a fragment substitution with the help of the simulator through the example described in Section 6.1.1.2 (see Fig.23). While defining the substitution *Insert new track*, the following process can be followed with the help of the simulator:

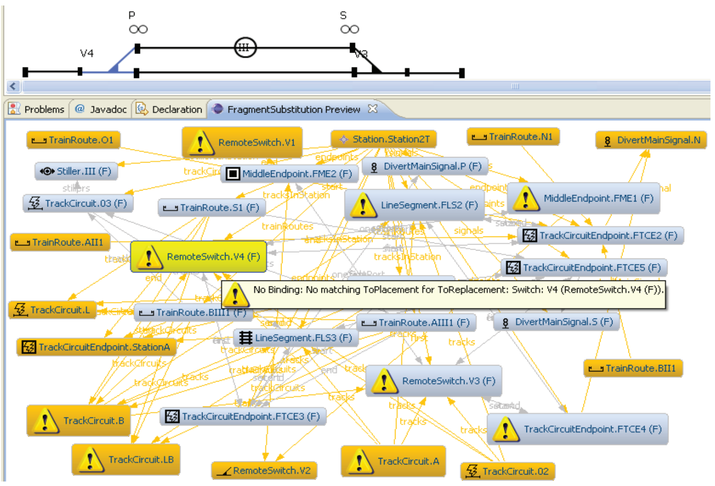
Run the simulation of the substitution with the default bindings. The default bindings are suggested automatically between any type-compatible boundary element. As part of the CVL methodology, we recommend the developer to start with inspecting and improving the default bindings instead of starting from scratch. By simulating the substitution with default bindings, the developer may obtain clues on how to improve the bindings based on the visualized simulation result.

Fig.24 gives a preview of the resulting model excerpt of *Insert new track* with default bindings. The rectangles representing the newly added elements (replacement fragment) are colored in blue. The dark yellow rectangles represent the elements that are directly related to the newly added elements in the resulting model excerpt.

Each rectangle consists of an icon which is the same one used in the base DSL graphical editor, the type of the element, and the name of the element. For example, in the rectangle representing the line segment *FLS3* (see Fig.24), the same icon is used in the TCL editor for line segments, followed by its type and name.

15 http://www.eclipse.org/gef/zest/

By selecting a rectangle, the color will turn to light yellow in the preview. At the same time, the actual model element in the base model or library models will be highlighted in the CVL-enabled base DSL editor. For example, as illustrated in Fig.24, by selecting a newly added element *RemoteSwitch.V4(F)*, its color changes in the preview, and the switch *V4* is highlighted in blue in the CVL-enabled TCL editor.



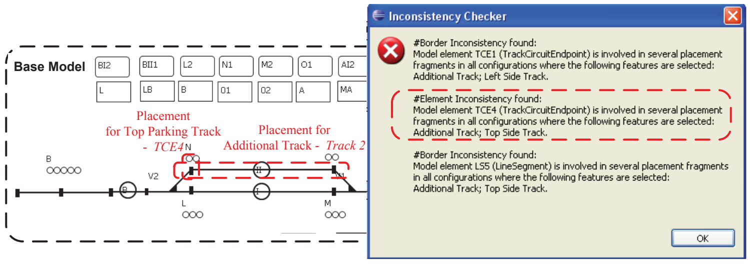
No Binding: No matching ToPlacement for ToReplacement: Switch: V4 (RemoteSwitch.V4 (F)).

Fig.24. Preview of the resulting model excerpt of the fragment substitution *Insert new track* with default bindings

In the preview all the elements with missing references (due to unbound boundary elements) will be marked with a warning sign. A warning message with the type information of the unbound reference also pops up when the element is selected in the preview, which should be inspected manually to rule out unintentional missing references. As illustrated in Fig.24, it is warned that a boundary element recording a reference to the remote switch *V4* is left unbound, which requires closer inspection from the developer to see whether this reference is unbound intentionally.

Run the simulation of the substitution with default bindings. In this step the developer improves the default bindings based on the preview of the simulation result. The simulator can be invoked iteratively on newer versions of the binding definition until the resulting model excerpt fulfills the intention of the developer.

6.2.2.2 Checking the Consistency between Variability Specification and Variability Realization


Base Model
BI2
BII1
L2
N1
M2
O1
AI2
L
LB
B
01
02
A
MA
Placement for Top Parking Track - TCE4
Placement for Additional Track - Track 2
B
N
V2
L
M
Inconsistency Checker
OK

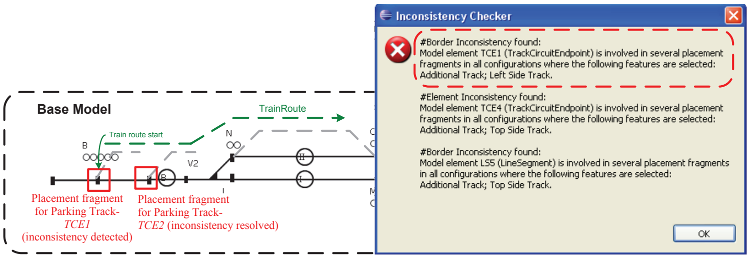
```
#Border Inconsistency found:
Model element TCE1 (TrackCircuitEndpoint) is involved in several placement
fragments in all configurations where the following features are selected:
Additional Track; Left Side Track.

#Element Inconsistency found:
Model element TCE4 (TrackCircuitEndpoint) is involved in several placement
fragments in all configurations where the following features are selected:
Additional Track; Top Side Track.

#Border Inconsistency found:
Model element LS5 (LineSegment) is involved in several placement fragments
in all configurations where the following features are selected:
Additional Track; Top Side Track.
```

Fig.25. Element inconsistency detected by the consistency checker

As stated in [21, 22], ensuring the consistency between variability specification and realization is a big underrated challenge in SPL development. Imagine that in a product line, the variability specification allows the coexistence of feature A and B in products, but the realization of feature A and B involves changing the same model object in two different ways. If this inconsistency between variability specification and realization is not detected and rectified at design time, it will cause errors during product derivation.


Base Model
TrainRoute
Train route start
B
N
V2
Placement fragment for Parking Track- TCE1 (inconsistency detected)
Placement fragment for Parking Track- TCE2 (inconsistency resolved)
Inconsistency Checker
OK

```
#Border Inconsistency found:
Model element TCE1 (TrackCircuitEndpoint) is involved in several placement
fragments in all configurations where the following features are selected:
Additional Track; Left Side Track.

#Element Inconsistency found:
Model element TCE4 (TrackCircuitEndpoint) is involved in several placement
fragments in all configurations where the following features are selected:
Additional Track; Top Side Track.

#Border Inconsistency found:
Model element LS5 (LineSegment) is involved in several placement fragments
in all configurations where the following features are selected:
Additional Track; Top Side Track.
```

Fig.26. Border inconsistency detected by the consistency checker

In order to address this challenge, we have developed a consistency checker based on CVL and Alloy [67] formal analysis (see Section 2.4). The consistency checker checks the consistency between the feature specification and product realization layer of a CVL model. The consistency checker is built to detect the following types of inconsistencies:

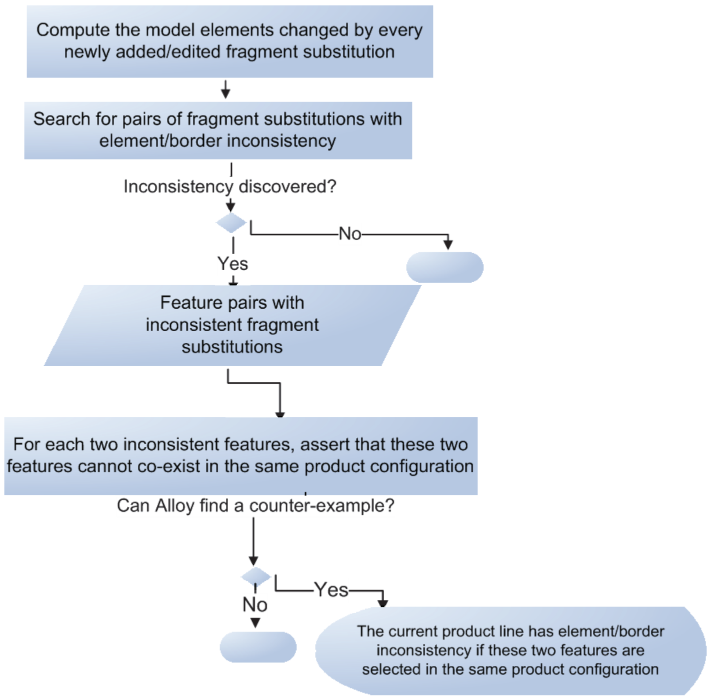

Compute the model elements changed by every newly added/edited fragment substitution
Search for pairs of fragment substitutions with element/border inconsistency
Inconsistency discovered?
No
Yes
Feature pairs with inconsistent fragment substitutions
For each two inconsistent features, assert that these two features cannot co-exist in the same product configuration
Can Alloy find a counter-example?
Yes
No
The current product line has element/border inconsistency if these two features are selected in the same product configuration

Fig.27. How the consistency checker works internally

Element inconsistency occurs when one model element in the base model is to be replaced in two substitutions, and both substitutions coexist in the same product configuration (resolution model) (see Fig.25). By executing a product configuration with element inconsistencies, the substitution executed later in time will overwrite the changes applied to the "conflicting" element by another substitution earlier.

Border inconsistency occurs when two model elements in the base model are directly connected, included in the placement fragments of two different substitutions, and both the substitutions are selected in the same product configuration (resolution model) (see Fig.26). By executing a product configuration with border inconsistencies, either the CVL transformation may halt if an exception is thrown, or the reference(s) at the "border" may be incorrectly set to *null* instead of the intended model element(s).

As illustrated in Fig.27, when the consistency checker is invoked, it first traverses every newly added/edited fragment substitution to search for pairs of substitutions with element/border inconsistencies. For each pair of fragment substitutions with inconsistencies, the consistency checker renders the alloy analyzer to see if the features that contain these two substitutions can coexist in any valid product configuration. If so, then there are element/border inconsistencies between the feature specification and realization layer of this CVL model. For resolving element/border inconsistencies, Anatoly has continued our work and suggested categorized solutions in [129].

6.3 RT3: Evolving Model-Driven SPLs

There is a lack of automatic assistance in supporting the evolution of model-driven SPLs. In Section 6.3.1, 6.3.2 and 6.3.3 we describe how our artifacts can contribute to RT3.1, RT3.2 and RT3.3.

6.3.1 RT3.1: How to improve the productivity in inducing SPL evolution steps from new requirements?

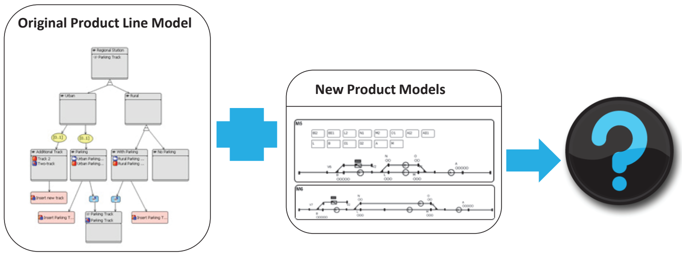


Fig.28. The scenario of augmenting a station product line with two new station products

When it comes to the evolution of product lines, augmenting an existing product line with new products is a typical scenario. As reported in [121] and [123], train experts from ABB, Norway specify station models and station product lines based on the station drawings received from Norwegian railway authorities. As illustrated in Fig.28, with newly received drawings, it would be useful for the train experts to understand if and how the new station products can fit into the existing product line. We learned from literature review that this has been mostly a manual process, and we identified the requirement for providing automatic assistance to the developer in product line augmentation (see RT3.1), which resulted in our 5th artifact ("Augmenting an SPL") (Appendix V).

In Paper V, we proposed an approach for augmenting product lines with new products. The approach has the following characteristics:

```
Input
•  PL:   a CVL model as the original product line
   model
•  newProducts = {P_1,P_2,...,P_n}   : a set of new product
   models specified in a MOF-based modeling
   language
1  For each $P_n \in newProducts$
2        augment ($PL$, $P_n$) assigns $PL$
3  End
Output PL: the augmented CVL model
```

Fig.29. Pseudo-code for the overview of the SPL augmentation algorithm

Generic. This approach is built based on CVL (the 1st artifact) and CVL Compare (the 3rd artifact). Therefore it can be applied to any CVL-based product line with any MOF-based base modeling language.

Automated. This approach provides an automatic procedure for product line augmentation. First it takes an existing product line model (a CVL model) and a set of new product models defined in the same base language as input, and then the following routines will be invoked to : (1) Check if the new products can already be derived from the existing product line model. (2) If not, augment the existing product line model with necessary variability realization accordingly. The output of the procedure will be a tentative augmented product line model, which can serve as a baseline for manual enhancement using other CVL tools.

As illustrated in Fig.29, the existing product line is augmented with one new product each time until all the new products are incorporated. We refer to Paper V (Appendix V) for a detailed description of the augmentation algorithm. In the following we give a simplified description of a single augmentation procedure (see Fig.30):

1) **Compare the base model of the current product line model *PL* with the new product model *P* using CVL Compare (the 3rd artifact).** The comparison will result in a CVL model ΔPL.
   a) If ΔPL does not contain any fragment substitution, then the new product model *P* is equivalent to the base model of *PL*. The new product *P* is already included in the current product line *PL*.
   b) If ΔPL does contain a fragment substitution ΔS *(ΔPF, ΔRF)* with ΔPF and ΔRF as its placement and replacement fragment, then the new product model *P* can be obtained by applying ΔS *(ΔPF, ΔRF)* to the base model of the current product line model. Subsequently we need to check if the current product line model *PL* already includes a fragment substitution *S(PF, RF)* that is equivalent to ΔS *(ΔPF, ΔRF)*, starting from Step 2).
2) **Search the current product line model *PL* to see if it includes an existing placement fragment *PF* that is equivalent to ΔPF.**
   a) If so, then proceed with Step 3).
   b) If not, then the new product *P* is not included in the current product line *PL*. The augmentation routine is invoked to add the placement fragment ΔPF, replacement fragment ΔRF and the new fragment substitution ΔS *(ΔPF, ΔRF)* to the current product line model.

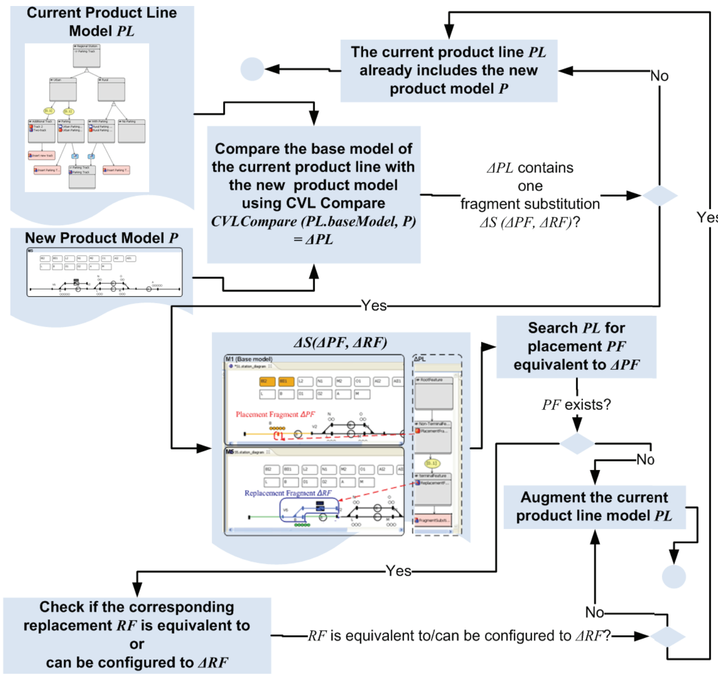

Current Product Line Model *PL*
New Product Model *P*
The current product line *PL* already includes the new product model *P*
No
Compare the base model of the current product line with the new product model using CVL Compare *CVLCompare (PL.baseModel, P)* = *ΔPL*
ΔPL contains one fragment substitution *ΔS (ΔPF, ΔRF)*?
Yes
Yes
ΔS(ΔPF, ΔRF)
Placement Fragment *ΔPF*
Replacement Fragment *ΔRF*
Search *PL* for placement *PF* equivalent to *ΔPF*
PF exists?
No
Augment the current product line model *PL*
Yes
Check if the corresponding replacement *RF* is equivalent to or can be configured to *ΔRF*
No
RF is equivalent to/can be configured to *ΔRF*?

Fig.30. The simplified procedure of augmenting an SPL with a new product

3) **Check if *PF*'s corresponding replacement fragment *RF* is equivalent to Δ*RF* or can be configured to Δ*RF*.** As described in Section 6.1.1.1, with CVL, the developer is capable of further customizing/configuring a replacement fragment by defining placements inside it. Therefore the routine of this step does not only check if *RF* is equivalent to Δ*RF*, but also check if *RF* can be configured to Δ*RF* through any placement inside it.
   a) If *PF*'s corresponding replacement fragment *RF* is equivalent to Δ*RF* or can be configured to Δ*RF*, then the current product line model *PL* already includes a fragment substitution *S(PF, RF)* that is equivalent to Δ*S* (Δ*PF*, Δ*RF)*. Therefore the current product line model *PL* can already derive the new product model *P* and does not need augmentation.
   b) If *PF*'s corresponding replacement fragment *RF* is neither equivalent to Δ*RF* nor can be configured to Δ*RF*, then the new product *P* is not included in the current product line *PL*. The augmentation routine is invoked to configure *RF* into Δ*RF* and add corresponding substitutions accordingly.

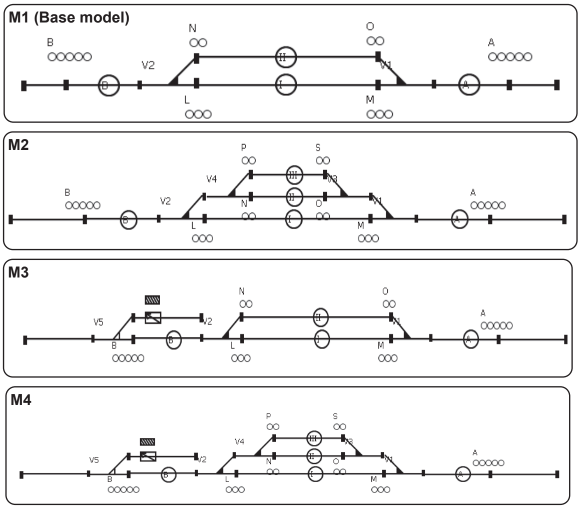


Fig.31. The four products of the original product line

As described above, the new product *P* is first compared with the base model of the current product line *PL* using CVL Compare. The resulting CVL model *ΔPL* contains only one fragment substitution *ΔS(ΔPF, ΔRF)*. This fragment substitution shows how to obtain the new product *P* from the base model by substituting *ΔPF* with *ΔRF*. However, the placement fragment *ΔPF* may suggest more than one variation point which we cannot identify by only comparing *P* with the base model without additional information.

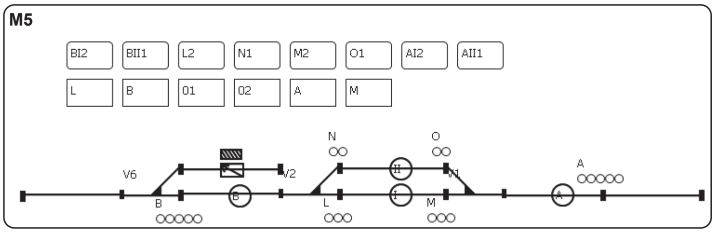


Fig.32. The new station *M5*

Suppose that we need to augment the station product line with another new station *M6* (see Fig.35) after *M5* (see Fig.32) is incorporated into the original product line (see Fig.31).

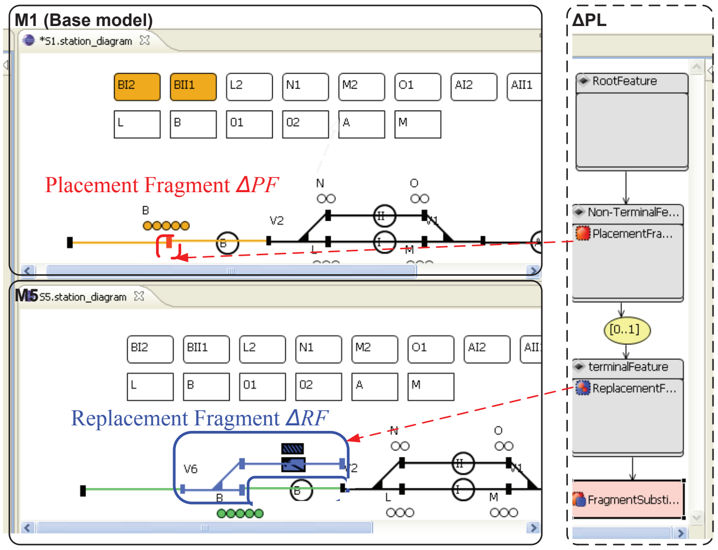


Fig.33. Compare *M5* with *M1* (base model) using CVL Compare

We compare *M1* (base model) with *M6* using CVL Compare. The resulting CVL model contains the fragment substitution *ΔS'(ΔPF', ΔRF')*. The placement fragment *ΔPF'* refers to a model fragment with disjoint parts *ΔPF*$_1$*'* and *ΔPF*$_2$*'* (annotated in *M1* in Fig.36). As annotated in *M6* in Fig.36, the replacement fragment *ΔRF'* also refers to a model fragment with disjoint parts *ΔRF*$_1$*'*, *ΔRF*$_2$*'* and *ΔRF*$_3$*'*.

We further search in the current product line (see Fig.34) for a node with the scope of *ΔPF'* and fail to find one. Thus *ΔPF'* will be regarded as a new variation point to be added into the current product line. However, we can notice that *ΔPF*$_1$*'*, as a subset of *ΔPF'*, coincides with *ΔPF* (see Fig.33). It would be ideal that the equivalence of *ΔPF*$_1$*'* and *ΔPF* is detected, so that *ΔPF'* will be regarded as two features with the scope of *ΔPF*$_1$*'* and *ΔPF*$_2$*'*. *ΔPF*$_1$*'* will then be synthesized into the current feature *Parking* (see Fig.34) with the scope of *ΔPF*. In addition, a new feature will be created with the scope of *ΔPF*$_2$*'*. Nevertheless, our current algorithm does not yet support cases like detecting and splitting a compound variability.

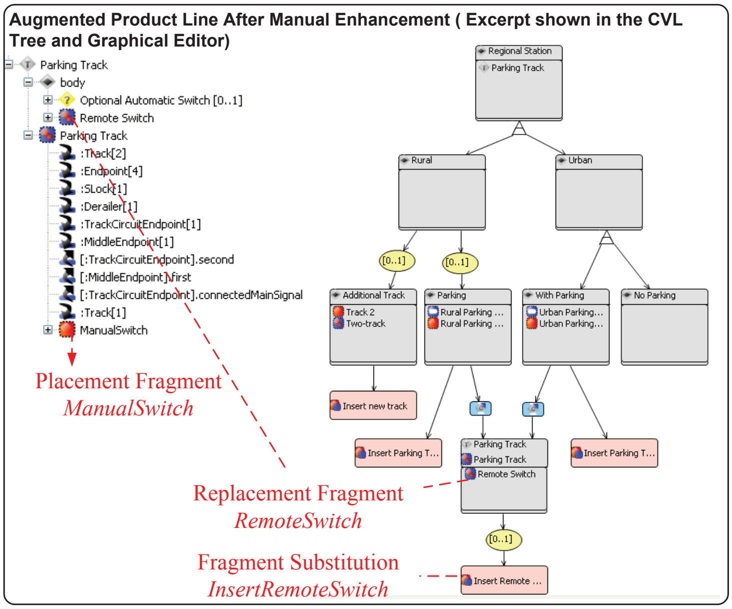


Fig.34. The product line model augmented by *M5* (after manual enhancement)

A possible solution to detect compound variability is to provide the developer with some automatic assistance, such as comparing placements/replacements, in synthesizing, refactoring and optimizing newly added features after all the new products are incorporated into the current product line model. Another possible solution is to apply some domain-specific semantic information of the product models to identify a compound variability.

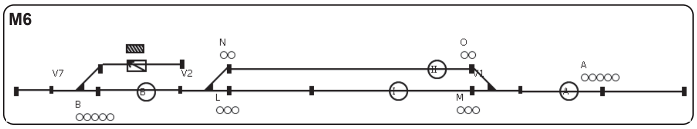


Fig.35. The new station *M6*

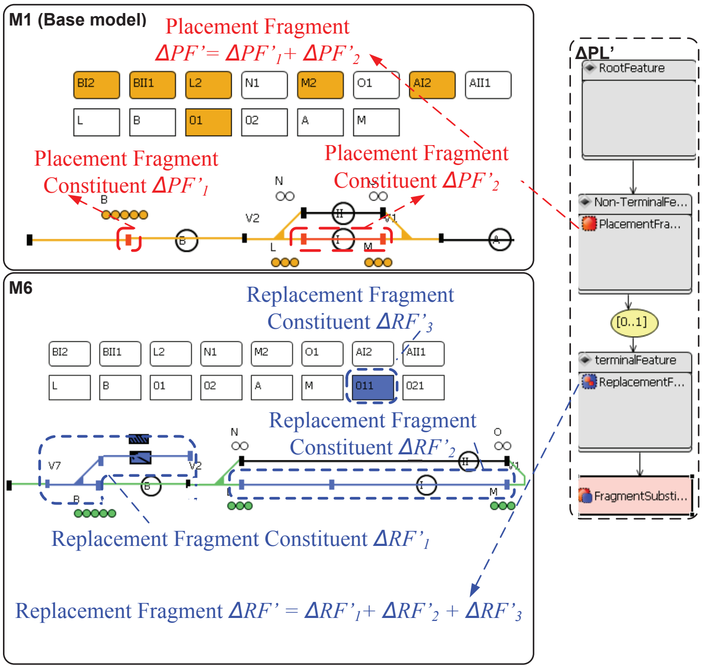

M1 (Base model)
Placement Fragment
ΔPF'= ΔPF'1+ ΔPF'2
BI2
BII1
L2
N1
M2
O1
AI2
AII1
L
B
01
02
A
M
Placement Fragment Constituent ΔPF'1
Placement Fragment Constituent ΔPF'2
N
V2
V1
L
M
B
A
M6
Replacement Fragment Constituent ΔRF'3
011
021
Replacement Fragment Constituent ΔRF'2
V7
O
Replacement Fragment Constituent ΔRF'1
Replacement Fragment ΔRF' = ΔRF'1+ ΔRF'2 + ΔRF'3
ΔPL'
RootFeature
Non-TerminalFe…
PlacementFra…
[0..1]
terminalFeature
ReplacementF…
FragmentSubsti…

Fig.36. Compare *M6* with *M1* (base model) using CVL Compare

6.3.2 RT3.2: How to improve the productivity of SPL co-evolution?

As discussed in RT3.2, for a model-driven SPL, the metamodel of the base DSL, the core assets (base & library model), the product line model and all the intended product models depend upon each other. All of them can subject to changes during product line evolution. Therefore, it can become necessary to co-evolve some others when some of them them evolves.

As illustrated in Fig.37, one of the common scenarios for SPL co-evolution occurs when the core assets of a product line are changed but its product derivation needs to remain unaffected. In order to ensure that, does the developer need to update the product line model? If so, how? We identified the requirement for a method for co-evolution of core assets and product line model, which resulted in our 6th artifact ("Co-evolving an SPL") (Appendix VI).

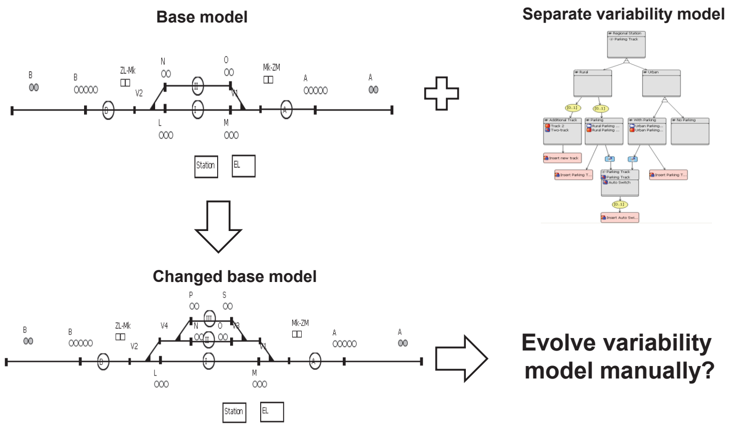

Base model
Separate variability model
Changed base model
Evolve variability model manually?

Fig.37. The scenario of evolving an SPL when the base model is changed

In Paper VI (Appendix VI), we proposed an approach for co-evolving the product line model (CVL model) when the base model is evolved. The approach has the following characteristics:

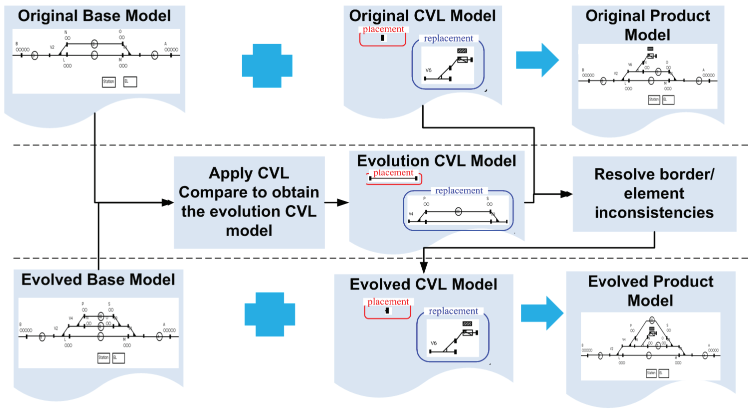


Fig.38. The automatic process of evolving an SPL when the base model is changed

Generic. The approach is built based on CVL (the 1st artifact) and CVL Compare (the 3rd artifact), and can be applied to any CVL-based product line and MOF-based base DSL.

Automated. The approach provides an automatic procedure, which takes the original base model, the evolved base model and the original CVL model as input.

With the input models, the following routines will be invoked to: (1) Check if the changes to the original base model will affect product derivation. (2) If not, then there is no need to co-evolve the original CVL model. (3) If so, then the changes to the original base model have caused element/border inconsistencies. The inconsistencies need to be resolved by co-evolving the original CVL model. Based on a combined analysis of the original CVL model and the changes to the original base model, an evolved CVL model will be induced and suggested to the developer. Note that the automatic procedure may require human intervention when it comes to non-deterministic decisions.

In the following we give a simplified description of the automatic procedure (see Fig.38). We refer to Paper VI (Appendix VI) for a detailed description of the approach.

1) **Apply CVL Compare to the original base model and the evolved base model.** This step results in a CVL model "the evolution CVL model". The evolved base model can be obtained by applying the only fragment substitution in the evolution CVL model to the original base model. As illustrated in Fig.38, a two-track station (the original base model) and a three-track station (the evolved base model) is compared using CVL Compare. The resulting evolution CVL model summarizes the changes that have been done to the original base model, which is to substitute the second track with a two-track structure.
2) **Check if there is any element/border inconsistency between the original and the evolution CVL model.** In this step the routine will check if there is any element/border inconsistency (see Section 6.2.2.2) between any placement fragment in the original CVL model and the only placement fragment in the evolution CVL model. As the example illustrated in Fig.38, the placement fragment in the evolution CVL model, which is the second track, starts with an endpoint on the left. The same endpoint is also included in a placement fragment in the original CVL model and these two placement fragments may exist in the same product, which causes an element inconsistency.
    a) If not, then the evolved base model will yield the same products as the original one, thus there is no need to co-evolve the original CVL model.
    b) If so, then proceed with Step 3) for inconsistency resolution.
3) **Resolve element/border inconsistencies**. The routine will perform the following to resolve an element/border inconsistency:
    a) Fetch the model elements from the replacement fragment (in the evolution CVL model) that are bound to the common endpoint E_1.
    b) Check if the model elements obtained from the last step have the same context (references to/from the same elements).
        i) If there is only one model element E_2 fulfills the requirement, then the element inconsistency can be automatically resolved. The resolution is performed by replacing the model element E_1 that causes the inconsistency by the model element E_2 in the original CVL model, which results in an evolved CVL model. In our example (see Fig.38), only the second left-most endpoint on the bottom track (from the replacement fragment in the evolution CVL model) has the same context as the common endpoint that causes the inconsistency.

ii) If there are more than model elements which fulfill the requirement (have the same type and context), then the developer will be asked to make the decision. In Paper VI, we also discussed the possibility to improve the automatic decision-making by taking the semantics of the base DSL into account.

6.3.3 RT3.3: How to assist the developer to gain a comprehensive understanding of the impact of an SPL evolution?

As discussed in RT 3.3, there is an emerging need for differencing techniques for product lines, which can be very useful for helping understand the impact of evolution by comparing the original and the evolved product line [6]. Most existing differencing techniques for product lines are dedicated to comparing feature models. The majority of those techniques are syntax-based, namely that they will only produce syntactical difference between the original and the evolved feature models [6]. On the other hand, the minority of existing feature model differencing techniques focus on semantic differencing, namely that the difference between the original and the evolved feature model will be represented in terms of added and removed products [6, 117]. However, semantic differencing techniques for feature models cannot always accommodate the need for differencing product lines, e.g., not feature model but variability (feature) realization has been changed during evolution. We identified the requirement for proposing a semantic differencing technique for product lines covering both variability specification and realization, which resulted in our 7th artifact ("Semantic differencing for SPLs") (Appendix VII).

In Paper VII (our 7th artifact), we proposed semantic differencing approach for product line. The approach aims to provide the developer with a comprehensive picture of the semantic impact of an evolution. Therefore it does not only take variability (feature) specification but also variability (feature) realization into account in the differencing process. We illustrated our approach on CVL-based product lines, which is one of the very few techniques that allow specifying both features and their realizations in a holistic product line model.

In particular, we proposed the following two semantic differencing operators, one for differencing the feature specification layer and the other for differencing the feature realization part of the original and the evolved product line model (CVL model):

***FSDiff* ("Differencing Feature Specification").** The semantic of the feature specification layer of a CVL model is all the valid configurations allowed by this layer, which we call "feature configurations". A feature configuration contains a set of features. Two feature configurations are regarded different if they contain different sets of features. As illustrated in Fig.39, by applying *FSDiff* to the original and the evolved product line *PL* and *PL'*, feature configurations that are newly added into the evolved product line will be output as a set of *FSDiff_Witnesses*. Each *FSDiff_Witness* represents a newly added feature configuration to the evolved product line. A diff CVL model *FSDIff_CVL* can be generated which only allows *FSDiff_Witnesses*.

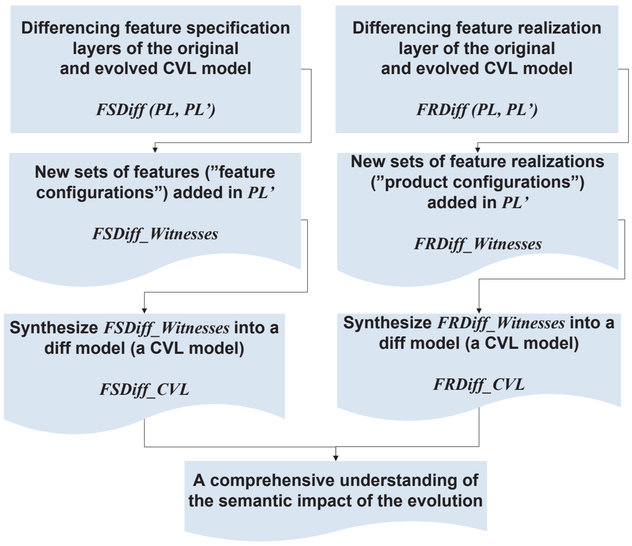

Differencing feature specification layers of the original and evolved CVL model
FSDiff (PL, PL')
Differencing feature realization layer of the original and evolved CVL model
FRDiff (PL, PL')
New sets of features ("feature configurations") added in *PL'*
FSDiff_Witnesses
New sets of feature realizations ("product configurations") added in *PL'*
FRDiff_Witnesses
Synthesize *FSDiff_Witnesses* into a diff model (a CVL model)
FSDiff_CVL
Synthesize *FRDiff_Witnesses* into a diff model (a CVL model)
FRDiff_CVL
A comprehensive understanding of the semantic impact of the evolution

Fig.39. Apply *FSDiff* and *FRDiff* to analyze the semantic impact of an evolution

***FRDiff* ("Differencing Feature Realization").** Feature realizations in a CVL model are usually contained by features. Therefore the semantic of the feature (product) realization layer of a CVL model is all the valid configurations allowed by both the feature specification and feature realization layer, which we call "product configurations". A product configuration contains a set of features and their realizations. Since only feature realizations are executed during product derivation, we only consider two product configurations different if they contain different sets of feature realizations. As illustrated in Fig.39, by applying *FRDiff* to the original and the evolved product line *PL* and *PL'*, product configurations that are newly added into the evolved product line will be output as a set of *FRDiff_Witnesses*. Each *FRDiff_Witness* represents a newly added product configuration to the evolved product line. A diff CVL model *FRDIff_CVL* can be generated which only allows *FRDiff_Witnesses*.

Formal definitions of *FSDiff* and *FRDiff* are given in Paper VII. In order to evaluate the feasibility of the approach, we also implemented the approach using Alloy (see Section 2.4). In particular, we performed the following tasks: (1) Define a simplified CVL metamodel using Alloy language. (2) Translate any CVL model into Alloy language. (3) Define *FSDiff* and *FRDiff* in Alloy language.

7 Discussions

In our thesis work we have addressed the research topics defined in Section 3 through artifacts manifested by our research papers. In this section we discuss how our artifacts can serve different roles in an extended CVL methodology for model-driven SPL development and evolution, which subsumes the methodology described in Section 6.1.1.2.

7.1 An Extended CVL Methodology for Model-Driven SPL Development and Evolution

In the following we walk through the extended CVL methodology as illustrated in Fig.40:

Develop the Base DSL
We observed at our industrial partners that in order to promote model-driven SPL development, we often needed to help them to adopt stand-alone model-driven software development first, namely that developing a base DSL with its tool support. Our 2nd artifact ("APRiL") is applicable in such contexts that, there is no existing DSL for the base domain of the SPL, or the current DSL is not suitable to serve as the base language for the SPL to be built.

During the development of a base DSL, the developer may benefit from the guidelines and lessons learned given in the 2nd artifact, such as keeping language definition simple, fully domain-specific without variability modeling concepts and etc.

Variability Identification
During variability identification, the developer needs to identify the variability and commonality of all intended products of the product line. We provide different strategies for identifying variability depending on the context of the SPL development:

(1) When the product line needs to include existing products, it would be suitable to compare existing products to identify variability between them.

- If existing product models specified in the base DSL are available, the developer can apply the CVL Compare tool provided in the 3rd artifact to synthesize a preliminary product line model from the existing product models automatically. Subsequently the preliminary product line model can be further enhanced by the developer manually by following the guidelines provided by the 1st artifact.

- If not, the developer can follow the step described in the 1st artifact to identify the variability of the SPL manually.

(2) When the product line is developed from scratch, the developer can start with choosing/specifying the base model. While defining the CVL model, the developer can then decide, how the intended product models should vary from each other relative to the base model.

(3) Very often a product line does not only need to include existing products, but also needs to introduce new products. In this context, the developer may need to apply a combination of the two strategies, by starting with comparing existing products, and further generalizing the product line to support new products.

Variability Specification
During variability specification, the developer needs to specify the high-level variability (domain-specific features) of the product line in the product line definition. Using the CVL language and tools provided by our 1st artifact, the developer can specify the high-level variability at the feature specification layer of a separate variability model, regardless of which base DSL is in use.

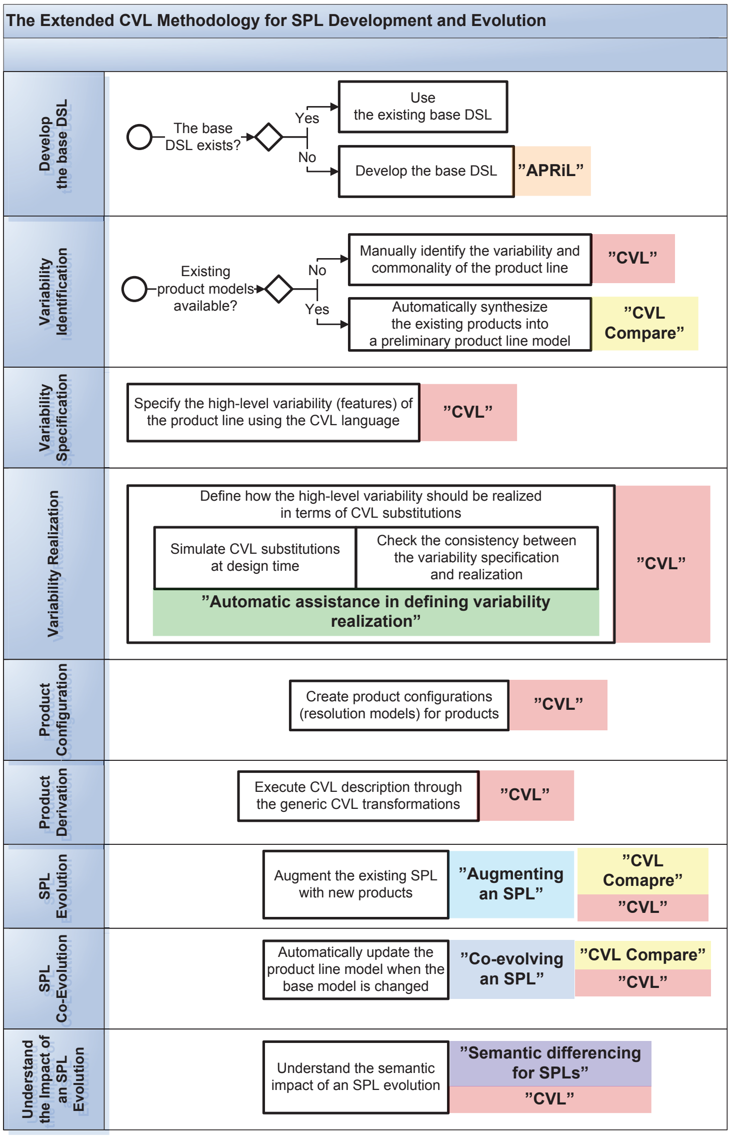
Variability Realization
During variability realization, the developer needs to define how the domain-level features of the product line should be realized at the model object level.

The CVL language (the 1st artifact) provides capabilities to define variability realization holistically with variability specification, by means of defining CVL-specific model editing model operations (substitutions) at the product realization layer of a CVL model.

In order to ensure that the product line model only yields intended product models, the developer can apply the two tools provided by our 4th artifact: the fragment substitution simulator to preview results of CVL substitutions, and the consistency checker to search for inconsistencies between the feature specification and product realization layer of a CVL model that may affect the intended product derivation.

Product Configuration
With the language and tool support provided by our 1st artifact, the developer can either use the standard CVL editor to create product configurations (resolution models in CVL terms), or render automatic tool support from the CVL "select (the features)-and-generate" resolution model generator.

Product Derivation
By running the CVL description through the generic CVL transformation (see the 1st artifact), product models can be derived from the product line model and the core assets of the product line.



The Extended CVL Methodology for SPL Development and Evolution
Develop the base DSL
The base DSL exists?
Yes
No
Use the existing base DSL
Develop the base DSL
"APRiL"
Variability Identification
Existing product models available?
No
Yes
Manually identify the variability and commonality of the product line
"CVL"
Automatically synthesize the existing products into a preliminary product line model
"CVL Compare"
Variability Specification
Specify the high-level variability (features) of the product line using the CVL language
"CVL"
Variability Realization
Define how the high-level variability should be realized in terms of CVL substitutions
Simulate CVL substitutions at design time
Check the consistency between the variability specification and realization
"Automatic assistance in defining variability realization"
"CVL"
Product Configuration
Create product configurations (resolution models) for products
"CVL"
Product Derivation
Execute CVL description through the generic CVL transformations
"CVL"
SPL Evolution
Augment the existing SPL with new products
"Augmenting an SPL"
"CVL Comapre"
"CVL"
SPL Co-Evolution
Automatically update the product line model when the base model is changed
"Co-evolving an SPL"
"CVL Compare"
"CVL"
Understand the Impact of an SPL Evolution
Understand the semantic impact of an SPL evolution
"Semantic differencing for SPLs"
"CVL"

Fig.40. The extended CVL methodology for Model-Driven SPL development and Evolution

SPL Evolution
Augmenting an existing product line with new products is a common SPL evolution requirement. The developer can choose to perform the augmentation by the following three means:

- Feed the automatic tool provided by our 5th artifact ("Augmenting an SPL") with the existing product line model and new product models. The tool will first check if the existing product line already includes the new products, and if not, the tool will update the product line model automatically. Subsequently the augmented product line model that is automatically suggested can be further enhanced by the developer manually by following the guidelines provided by the 1st artifact.
- Feed the new product models and all product models to the CVL Compare tool provided by our 3rd artifact. The CVL Compare will automatically synthesize an augmented product line model which includes all those products. However, it is not always practical to obtain all the product models of the existing product model.
- Follow the guidelines provided by our 1st artifact and augment the existing product line model manually.

SPL Co-Evolution
When the base model of a product line changes, the developer may need to update the product line model so that the current product derivation is not affected. In this context, the developer can apply the tool support provided by our 6th artifact, which will perform an automatic update of the product line model if necessary.

Understand the Impact of an SPL Evolution
In order to gain a comprehensive understanding of the semantic impact of an SPL evolution, the developer can use the two semantic differencing operators for product lines that we propose in the 7th artifact, one for differencing the feature specification layer of the original and the evolved CVL model, the other one for comparing the product realization layer of them.

8 Conclusions and Future Work

In our thesis work we have addressed the research topics defined in Section 3 through artifacts reflected by our research papers. In the following we revisit the research topics, which we conclude to what extent our artifacts have contributed to them, and propose potential directions for future work.

8.1 Research Topic 1: A Generic Approach for Developing Executable Model-Driven SPLs (RT1)

RT1 addressed how to develop executable model-driven SPLs in a generic way. This was decomposed into two sub-topics: how to define variability specification and realization in a generic way for model-driven SPLs in different domains (RT1.1), and how to develop a base DSL suited for building model-driven SPLs (RT1.2).

8.1.1 RT1.1: How to Define Variability Specification and Realization in a Generic Way for Model-Driven SPLs in Different Domains?

The 1st ("CVL"), 3rd ("CVL Compare") and 4th ("Automatic assistance in defining variability realization") artifact contribute to solving the research topic RT1.1 wherein we provide a separate and generic variability modeling language, the Common Variability Language (CVL), the CVL methodology and automatic support for model-driven SPL development.

While there are already several generic model-driven approaches for specifying domain-level variability in a separate product line model (e.g., feature modeling), few of them also support defining how the variability/features should be realized at the model (object) level in terms of model editing operations in the same product line model. The CVL approach is innovative in this aspect by introducing an additional product realization layer into the product line model. Therefore the developer can define both features and their realizations holistically in a single product line model which is fully executable.

Another key challenge was how to represent feature realization (model editing operations) in a generic way, such that it can describe arbitrary edits to any model specified in any base DSL. This is addressed in our CVL approach by generalizing arbitrary model editing operations into value, reference and fragment substitutions, which can describe arbitrary value and structural changes in any model.

Furthermore, the CVL methodology for SPL development provides guidelines on how the developer can manually develop a CVL-based SPL from scratch, and at which phases automatic support can be rendered to improve the productivity.

In summary, our artifacts have directly contributed to RT1.1 and the results have been validated through industrial and academia case studies. In order to contribute to RT1.1 further, the following directions can be explored in the future work of our proposed approach: (1) Propose more guidelines for CVL-based SPL development based on rigorous empirical studies. For example, as described in Section 6.1.1.2, the developer needs to make informed decisions in choosing the optimal base/library model for a product line, and the current descriptive guidelines can benefit from an update of quantitative metrics. (2) Propose new automatic support to cover all phases of CVL-based SPL development and update the guidelines accordingly, e.g., providing automatic support in choosing the optimal base/library model.

8.1.2 RT1.2: How to Develop a Base DSL Suited for Building Model-Driven SPLs?

The context of RT1.2 is to develop model-driven SPLs for a base domain when there is no base DSL yet. We observed at our industrial partners that instead of promoting model-driven SPL development to them alone, very often we also needed to help them to develop a base DSL first.

Our research design was decided by the nature of the challenge posed by RT1.2. Instead of trying to propose a new approach, we focus our research on developing base DSLs & model-driven SPLs for real industrial domains and summarizing guidelines based on the lessons learned. In paper II, we reported our experience in developing a base DSL and SPLs for the payroll reporting domain. Through the report, we showed: (1) It can be potentially beneficial if the language concepts of the base DSL is fully domain-specific without any variability modeling concept. This characteristic of the base DSL will make it more intuitive and conceptually clearer to build SPLs that are based on separate variability modeling approaches. (2) How the productivity of software development can be improved by the DSM paradigm and then even further elevated by model-driven SPL development.

In summary, our artifact has contributed to RT1.2 directly considering the empirical nature of this research topic. In order to strength the external validity of our study, the practice of "DSM+SPL" needs to be introduced to various industrial domains for empirical analysis from a broader spectrum.

8.2 Research Topic 2: Automatic Assistance in Model-Driven SPL Development (RT2)

RT2 addressed providing automatic assistance in model-driven SPL development. In particular, we focus on two sub-topics: how to improve the productivity of variability identification in model-driven SPL development (RT2.1), and how to ensure that variability realization will only yield intended products (RT2.2).

8.2.1 RT2.1: How to Improve the Productivity of Variability Identification in Model-Driven SPL Development?

During the variability identification phase of SPL development, the variability and commonality of all intended products of the product line need to be identified, which has been mostly a manual and time-consuming process. Instead of addressing RT2.1 for SPL development in all contexts, we narrowed our problem area down to a more specific but still quite common niche, which is - how to improve the productivity of variability identification when an SPL needs to include some existing products. Often in practice an SPL is not built from scratch. The product line needs to derive existing products, and on top of that, may further introduce new products. In this context, our CVL Compare approach (the 3rd artifact, together with the 1st one), provides automatic assistance to improve the productivity of variability identification.

In particular, the CVL Compare approach provides an automatic procedure to synthesize a CVL-based SPL from a set of existing product models defined in any MOF-based DSL. The CVL Compare tool takes in a set of existing product models, identifies the variability among them, and then outputs a preliminary product line model for manual enhancement.

In summary, the CVL Compare approach has partially contributed to RT2.1. However, the following aspects can be explored in future work: (1) The current approach can already induce simple feature dependencies such as co-existence. It will be useful to further improve the approach so that it can identify more complex feature dependencies and constraints based on the comparisons. (2) The current approach suggests the preliminary product line model that is rather flat. This is due to the lack of domain-specific semantic information in the generic approach. Potential extensions can be explored to allow human intervention in separating single features from compound features [142].

8.2.2 RT2.2: How to Ensure that Variability Realization will Only Yield Intended Products?

Defining variability realization is a critical step of an SPL development since it directly affects how the derived products will look like. Therefore the variability realization layer of a product line model (CVL model) should only yield intended products (RT2.2). However, it can be rather challenging to define variability realization when it involves much model (object) level details in a complex base domain. In the thesis work, we narrowed down the big problem into two specific areas: one is to provide immediate feedback on the definition of variability realization at design time; the other is to check the consistency between the variability realization and specification in the same product line model.

Firstly we proposed a fragment substitution simulator, which can be executed at design time and provide a preview of the resulting model excerpt. The simulator, if properly used in an iterative "define-preview-improve" manner, can provide an immediate feedback on whether the current definition of substitution will yield intended model changes. Secondly we proposed a consistency checker to search for unwanted element/border inconsistencies in the product line model, which may halt the product derivation or yield unintended products.

In summary, our approach has contributed directly to RT2.2 in two sub-areas. Future work can be performed in the following directions: (1) Applying pairwise testing techniques [69] to improve the scalability of the consistency checker. (2) Extending the consistency checker so that it can also verify additional rules of the base DSL (e.g., OCL constraints). (3) Suggest improvements to the product line model automatically from both the simulator and consistency checker.

8.3 Research Topic 3: Evolving Model-driven SPLs (RT3)

RT3 addressed approaches in evolving model-driven SPLs. In particular, we focused on three topics: how to improve the productivity in inducing SPL evolution steps from new requirements (RT3.1), how to improve the productivity of SPL co-evolution (RT3.2), and how to assist the developer to gain a comprehensive understanding of the impact of an SPL evolution (RT3.3).

8.3.1 RT3.1: How to Improve the Productivity in Inducing SPL Evolution Steps from New Requirements?

We narrowed down the scope of the problem to the following specific scenario: how to improve the productivity in augmenting an SPL when there is a need to include new products? We addressed this problem by providing an automatic procedure to augment a CVL-based product line with new product models. The approach takes in the current CVL model and the new product models as input, and outputs a tentative augmented CVL model for manual enhancement.

In summary, the proposed approach has directly contributed to a sub-area of RT3.1. However, we also observed an issue during the development of this approach, which is the identification compound variability as described in Section 6.3.1. The approach can benefit from future work in the following directions: (1) Evaluation using different examples in various domains. (2) The optimal strategy to identify compound variability. Can it be automated with comparison assistance? Can it be automated if the semantics of the base language of the product line is taken into account? (2) How to update feature dependencies and constraints based on the comparisons in the augmented product line model.

8.3.2 RT3.2: How to Improve the Productivity of SPL Co-evolution?

We narrowed down the scope of the problem to one of the most common SPL co-evolution scenario: evolving an SPL when the base model is changed. In order to address the challenge, we provided an approach to evolve the product line model automatically if necessary when the base model is changed. In particular, the approach applies CVL to record the changes to the base model, detect and resolve the element/boundary inconsistencies in the original product line model caused by the changes to the base model.

However, the automatically suggested evolved product line model with inconsistencies resolved is only syntactically correct and can be semantically invalid. This is because our approach cannot obtain the semantic information of the base DSL due to its generic nature. In potential extensions of this approach, we can consider to allow human intervention for decisions that cannot be made deterministically only based on the syntax of the base DSL. Furthermore, we may also focus on another common SPL co-evolution scenario, evolving the product line when the base DSL is changed, in our future work.

8.3.3 RT3.3: How to Assist the Developer to Gain a Comprehensive Understanding of the Impact of an SPL Evolution?

As discussed in Section 3.3, it is essential for the developer to gain a comprehensive understanding on the added/removed products after an SPL evolution, which calls for semantic differencing approaches for SPLs. We also saw that the traditional semantic differencing techniques for feature models cannot cater the need to differencing SPLs where the variability realization should also be taken in to account. In order to address this challenge, we proposed a semantic differencing approach for CVL-based SPLs which take both variability specification and variability realization into account. In particular, we defined and implemented (in Alloy) two semantic differencing operators for comparing the feature specification layer and the product realization layer of two CVL models. We illustrated the application on industrial case studies.

In summary, our approach has directly contributed to helping the developer to gain a comprehensive overview of the semantic impact of an SPL evolution. Furthermore, it can be extended in the following directions: (1) Improve the performance of the analysis by applying Alloy optimization techniques. (2) Integrate the approach with syntax-based differencing techniques for better performance and an even more complete understanding of an SPL evolution.

Bibliography

1. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P.: Extraction and Evolution of Architectural Variability Models in Plugin-Based Systems. Software &amp; Systems Modeling. 1-28 (2013)

2. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., and Lahire, P., "On Extracting Feature Models from Product Descriptions," Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, pp. 45-54, (2012)

3. Acher, M., Collet, P., Gaignard, A., Lahire, P., Montagnat, J., and France, R.B.: Composing Multiple Variability Artifacts to Assemble Coherent Workflows. Software Quality Control. 20, 689-734 (2012)

4. Acher, M., Collet, P., Lahire, P., and France, R., "Composing Feature Models," Proceedings of the Second international conference on Software Language Engineering, pp. 62-81, (2010)

5. Acher, M., Collet, P., Lahire, P., and France, R.: Managing Multiple Software Product Lines Using Merging Techniques. France: UniversityofNiceSophiaAntipolis. TechnicalReport, ISRN I3S/RR. 6 (2010)

6. Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., and Merle, P., "Feature Model Differences," Proceedings of the 24th international conference on Advanced Information Systems Engineering, pp. 629-645, (2012)

7. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C., "Refactoring Product Lines," Proceedings of the 5th international conference on Generative programming and component engineering, pp. 201-210, (2006)

8. Alves, V., Matos, P., Cole, L., Borba, P., and Ramalho, G., "Extracting and Evolving Mobile Games Product Lines," Proceedings of the 9th international conference on Software Product Lines, pp. 70-81, (2005)

9. Andersen, N., Czarnecki, K., She, S., and Wąsowski, A., "Efficient Synthesis of Feature Models," Proceedings of the 16th International Software Product Line Conference - Volume 1, pp. 106-115, (2012)

10. Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D.: Evaluating the "Small Scope Hypothesis". Unpublished. (2003)

11. Anfurrutia, F.I., Diaz, O., and Trujillo, S.: On Refining Xml Artifacts. In: Web Engineering, pp. 473-478. Springer (2007)

12. Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., and Sousa, A.: A Model-Driven Traceability Framework for Software Product Lines. Software &amp; Systems Modeling. 9, 427-451 (2010)

13. Apel, S., Kastner, C., and Lengauer, C., "Featurehouse: Language-Independent, Automated Software Composition," Proceedings of the 31st International Conference on Software Engineering, pp. 221-231, (2009)

14. Apel, S. and Kästner, C.: An Overview of Feature-Oriented Software Development. Journal of Object Technology. 8, 49-84 (2009)

15. Apel, S., Leich, T., Rosenmüller, M., and Saake, G., "Featurec++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," Proceedings of the 4th international conference on Generative Programming and Component Engineering, pp. 125-140, (2005)

16. Apel, S., Lengauer, C., Batory, D., Möller, B., and Kästner, C.: An Algebra for Feature-Oriented Software Development. University of Passau, MIP-0706. (2007)

17. Apel, S., Lengauer, C., Möller, B., and Kästner, C., "An Algebra for Features and Feature Composition," Proceedings of the 12th international conference on Algebraic Methodology and Software Technology, pp. 36-50, (2008)

18. Atkinson, C., Bayer, J., and Muthig, D.: Component-Based Product Line Development: The Kobra Approach. In: Software Product Lines, pp. 289-309. Springer (2000)

19. Babbie, E.R. and others: Survey Research Methods. Wadsworth Publishing Company Belmont, CA, (1990)

20. Batory, D., "Feature Models, Grammars, and Propositional Formulas," Proceedings of the 9th international conference on Software Product Lines, pp. 7-20, (2005)

21. Batory, D., Sarvela, J.N., and Rauschmayer, A.: Scaling Step-Wise Refinement. Software Engineering, IEEE Transactions on. 30, 355-371 (2004)

22. Batory, D. and Thaker, S.: Towards Safe Composition of Product Lines. Computer Science Department, University of Texas at Austin, (2006)

23. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M., "Pulse: A Methodology to Develop Software Product Lines," Proceedings of the 1999 symposium on Software reusability, pp. 122-131, (1999)

24. Benavides, D., Segura, S., and Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Inf. Syst. 35, 615-636 (2010)

25. Borba, P., Teixeira, L., and Gheyi, R.: A Theory of Software Product Line Refinement. Theoretical Computer Science. 455, 2-30 (2012)

26. Bosch, J., "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization," Proceedings of the Second International Conference on Software Product Lines, pp. 257-271, (2002)

27. Bosch, J. and Bengtsson, P., "Component Evolution in Product-Line Architectures," Proceedings of International Workshop on Component Based Software Engineering, (1999)

28. Bosch, J. and Capilla, R.: Variability Implementation. In: Systems and Software Variability Management, pp. 75-86. Springer (2013)

29. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., and Pohl, K.: Variability Issues in Software Product Lines. In: Software Product-Family Engineering, pp. 13-21. Springer (2002)

30. Bragança, A. and Machado, R.J., "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines," Software Product Line Conference, 2007. SPLC 2007. 11th International, pp. 3-12, (2007)

31. Bragança, A.M.T.: Methodological Approaches and Techniques for Model Driven Development of Software Product Lines. (2008)

32. Brooks Jr, F.P.: No Silver Bullet-Essence and Accidents of Software Engineering. IEEE computer. 20, 10-19 (1987)

33. Brun, C. and Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional. 9, 29-34 (2008)

34. Cengarle, M.V., Grönniger, H., and Rumpe, B., "Variability within Modeling Language Definitions," Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, pp. 670-684, (2009)

35. Chastek, G., Donohoe, P., Kang, K.C., and Thiel, S., "Product Line Analysis: A Practical Introduction," (2001)

36. Chen, K., Zhang, W., Zhao, H., and Mei, H., "An Approach to Constructing Feature Models Based on Requirements Clustering," Proceedings of the 13th IEEE International Conference on Requirements Engineering, pp. 31-40, (2005)

37. Clarke, D., Helvensteijn, M., and Schaefer, I., "Abstract Delta Modeling," ACM Sigplan Notices, pp. 13-22, (2010)

38. Clarke, E.M. and Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys (CSUR). 28, 626-643 (1996)

39. Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A., "Symbolic Model Checking of Software Product Lines," Proceedings of the 33rd International Conference on Software Engineering, pp. 321-330, (2011)

40. Clements, P. and Northrop, L.: Software Product Lines. Addison-Wesley Boston, (2002)

41. Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A., "Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features," Proceedings of the 2013 International Conference on Software Engineering, pp. 472-481, (2013)

42. Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., and Pietroszek, K., "Model-Driven Software Product Lines," Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 126-127, (2005)

43. Czarnecki, K. and Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., (2000)

44. Czarnecki, K. and Helsen, S.: Feature Modeling. Generative Programming. 82-130 (1998)

45. Czarnecki, K., Helsen, S., and Eisenecker, U.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice. 10, 143-169 (2005)

46. Czarnecki, K. and Pietroszek, K., "Verifying Feature-Based Model Templates against Well-Formedness Ocl Constraints," Proceedings of the 5th international conference on Generative programming and component engineering, pp. 211-220, (2006)

47. Czarnecki, K. and Wasowski, A., "Feature Diagrams and Logics: There and Back Again," Proceedings of the11th International Software Product Line Conference, pp. 23-34, (2007), (2007)

48. Deng, G., Lenz, G., and Schmidt, D.C., "Addressing Domain Evolution Challenges in Model-Driven Software Product-Line Architectures," Proceedings of the ACE/MODELS 2005 workshop on MDD for Software Product-Line Architectures, Jamaica, (2005)

49. Dhungana, D., Neumayer, T., Grunbacher, P., and Rabiser, R., "Supporting Evolution in Model-Based Product Line Engineering," Proceedings of the 2008 12th International Software Product Line Conference, pp. 319-328, (2008)

50. Dietrich, C., Tartler, R., Schröder-Preikschat, W., and Lohmann, D., "A Robust Approach for Variability Extraction from the Linux Build System," Proceedings of the 16th International Software Product Line Conference - Volume 1, pp. 21-30, (2012)

51. Dumitru, H., Gibiec, M., Hariri, N., Cleland-Huang, J., Mobasher, B., Castro-Herrera, C., and Mirakhorli, M., "On-Demand Feature Recommendations Derived from Mining Public Product Descriptions," Proceedings of the 33rd International Conference on Software Engineering, pp. 181-190, (2011)

52. Elsner, C., Botterweck, G., Lohmann, D., and Schröder-Preikschat, W., "Variability in Time - Product Line Variability and Evolution Revisited," VaMoS, pp. 131-137, (2010)

53. Fahrenberg, U., Legay, A., and Wasowski, A., "Vision Paper: Make a Difference! (Semantically)," Proceedings of the 14th international conference on Model driven engineering languages and systems, pp. 490-500, (2011)

54. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K., Svendsen, A., and Zhang, X.: A Generic Language and Tool for Variability Modeling. SINTEF, Oslo. (2009)

55. Fontoura, M., Pree, W., and Rumpe, B.: The Uml Profile for Framework Architectures. Addison-Wesley Longman Publishing Co., Inc., (2000)

56. Fritsch, C. and Hahn, R.: Product Line Potential Analysis. In: Software Product Lines, pp. 228-237. Springer (2004)

57. Ghanam, Y. and Maurer, F., "Linking Feature Models to Code Artifacts Using Executable Acceptance Tests," Proceedings of the 14th international conference on Software product lines: going beyond, pp. 211-225, (2010)

58. Gheyi, R., Massoni, T., and Borba, P., "A Theory for Feature Models in Alloy," First alloy workshop, pp. 71-80, (2006)

59. Gomaa, H.: Designing Software Product Lines with Uml. Addison-Wesley Boston, USA;, (2004)

60. Gomaa, H. and Saleh, M., "Software Product Line Engineering for Web Services and Uml," Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications, pp. 110-vii, (2005)

61. Griss, M.L., Favaro, J., and d'Alessandro, M., "Integrating Feature Modeling with the Rseb," Proceedings of the Fifth International Conference on Software Reuse, pp. 76-85, (1998), (1998)

62. Gruler, A., Leucker, M., and Scheidemann, K., "Calculating and Modeling Common Parts of Software Product Lines," Proceedings of the 2008 12th International Software Product Line Conference, pp. 203-212, (2008)

63. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Rumpe, B., Müller, K., and Schaefer, I., "Engineering Delta Modeling Languages," Proceedings of the 17th International Software Product Line Conference, pp. 22-31, (2013)

64. Haber, A., Kutz, T., Rendel, H., Rumpe, B., and Schaefer, I., "Delta-Oriented Architectural Variability Using Monticore," Proceedings of the 5th European Conference on Software Architecture: Companion Volume, pp. 6-6, (2011)

65. Haber, A., Rendel, H., Rumpe, B., and Schaefer, I., "Delta Modeling for Software Architectures," MBEES, pp. 1-10, (2011)

66. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., and Svendsen, A., "Adding Standardized Variability to Domain Specific Languages," Proceedings of the 2008 12th International Software Product Line Conference, pp. 139-148, (2008)

67. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. Methodol. 11, 256-290 (2002)

68. Jirapanthong, W. and Zisman, A.: Xtraque: Traceability for Product Line Systems. Software &amp; Systems Modeling. 8, 117-144 (2009)

69. Johansen, M.F., Haugen, O., Fleurey, F., Eldegard, A.G., and Syversen, T., "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines," Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems, pp. 269-284, (2012)

70. John, I. and Eisenbarth, M., “A Decade of Scoping: A Survey,” Proceedings of the 13th International Software Product Line Conference, pp. 31-40, (2009)

71. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S., “Feature-Oriented Domain Analysis (Foda) Feasibility Study,” (1990)

72. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M.: Form: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering. 5, 143-168 (1998)

73. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S., “Design Guidelines for Domain Specific Languages,” The 9th OOPSLA workshop on domain-specific modeling, pp. 7-13, (2009)

74. Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S., “Featureide: A Tool Framework for Feature-Oriented Software Development,” Proceedings of the 31st International Conference on Software Engineering, pp. 611-614, (2009)

75. Kelly, S. and Pohjonen, R.: Worst Practices for Domain-Specific Modeling. Software, IEEE. 26, 22-29 (2009)

76. Kelly, S. and Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley. com, (2008)

77. Kishi, T., Noda, N., and Katayama, T., “A Method for Product Line Scoping Based on a Decision-Making Framework,” Proceedings of the Second International Conference on Software Product Lines, pp. 348-365, (2002)

78. Kästner, C., Apel, S., and Kuhlemann, M., “Granularity in Software Product Lines,” Proceedings of the 30th international conference on Software engineering, pp. 311-320, (2008)

79. Kästner, C., Apel, S., Thüm, T., and Saake, G.: Type Checking Annotation-Based Product Lines. ACM Trans. Softw. Eng. Methodol. 21, 14:1-14:39 (2012)

80. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., and Jézéquel, J.-M.: Introducing Variability into Aspect-Oriented Modeling Approaches. In: Model Driven Engineering Languages and Systems, pp. 498-513. Springer (2007)

81. Liu, J. and Batory, D., “Automatic Remodularization and Optimized Synthesis of Product-Families,” Generative Programming and Component Engineering, pp. 379-395, (2004)

82. Lopez-Herrejon, R.E., “Language and Uml Support for Features: Two Research Challenges,” VaMoS, pp. 97-100, (2007)

83. Lopez-Herrejon, R.E. and Egyed, A., “Towards Fixing Inconsistencies in Models with Variability,” Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, pp. 93-100, (2012)

84. Lopez-Herrejon, R.E., Egyed, A., Trujillo, S., de Sosa, J., and Azanza, M., "Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering," VaMoS, pp. 93-100, (2010)

85. Lora-Michiels, A., Salinesi, C., and Mazo, R., "A Method Based on Association Rules to Construct Product Line Models," VaMoS, pp. 147-150, (2010)

86. Maoz, S., Ringert, J.O., and Rumpe, B., "Addiff: Semantic Differencing for Activity Diagrams," Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 179-189, (2011)

87. Maoz, S., Ringert, J.O., and Rumpe, B., "Cddiff: Semantic Differencing for Class Diagrams," Proceedings of the 25th European conference on Object-oriented programming, pp. 230-254, (2011)

88. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., and Saval, G., "Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis," Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, pp. 243-253, (2007)

89. Mitschke, R. and Eichberg, M., "Supporting the Evolution of Software Product Lines," ECMDA Traceability Workshop (ECMDA-TW), pp. 87-96, (2008)

90. Mohalik, S., Ramesh, S., Millo, J.-V., Krishna, S.N., and Narwane, G.K., "Tracing Spls Precisely and Efficiently," Proceedings of the 16th International Software Product Line Conference - Volume 1, pp. 186-195, (2012)

91. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M., "Weaving Variability into Domain Metamodels," Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, pp. 690-705, (2009)

92. Mussbacher, G., Whittle, J., and Amyot, D., "Semantic-Based Interaction Detection in Aspect-Oriented Scenarios," Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE, pp. 203-212, (2009)

93. Neves, L., Teixeira, L., Sena, D., Alves, V., Kulezsa, U., and Borba, P., "Investigating the Safe Evolution of Software Product Lines," ACM SIGPLAN Notices, pp. 33-42, (2011)

94. Niu, N. and Easterbrook, S., "On-Demand Cluster Analysis for Product Line Functional Requirements," Proceedings of the 2008 12th International Software Product Line Conference, pp. 87-96, (2008)

95. Noor, M.A., Rabiser, R., and Grünbacher, P.: Agile Product Line Planning: A Collaborative Approach and a Case Study. Journal of Systems and Software. 81, 868-882 (2008)

96. Owre, S., Rushby, J.M., Shankar, N., and Stringer-Calvert, D.W.J.: Pvs: An Experience Report. In: Applied Formal Methods—Fm-Trends 98, pp. 338-345. Springer (1999)

97. Paige, R.F., Brooke, P.J., and Ostroff, J.S.: Metamodel-Based Model Conformance and Multiview Consistency Checking. ACM Transactions on Software Engineering and Methodology (TOSEM). 16, 11-11 (2007)

98. Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., and Guo, J., “Feature-Oriented Software Evolution,” Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, pp. 17:1-17:8, (2013)

99. Pohl, K., Böckle, G., and Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, (2005)

100. Pree, W., Fontoura, M., and Rumpe, B., “Product Line Annotations with Uml-F,” Proceedings of the Second International Conference on Software Product Lines, pp. 188-197, (2002)

101. Reder, A. and Egyed, A., “Incremental Consistency Checking for Complex Design Rules and Larger Model Changes,” Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems, pp. 202-218, (2012)

102. Ryssel, U., Ploennigs, J., and Kabitzsch, K., “Extraction of Feature Models from Formal Contexts,” Proceedings of the 15th International Software Product Line Conference, Volume 2, pp. 4:1-4:8, (2011)

103. Ryssel, U., Ploennigs, J., and Kabitzsch, K., “Reasoning of Feature Models from Derived Features,” Proceedings of the 11th International Conference on Generative Programming and Component Engineering, pp. 21-30, (2012)

104. Sampaio, A., Rashid, A., Chitchyan, R., and Rayson, P.: Transactions on Aspect-Oriented Software Development Iii. In, A. Rashid and M. Aksit, (eds.), pp. 4-39. Springer-Verlag (2007)

105. Satyananda, T.K., Lee, D., and Kang, S., “Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line,” Proceedings of the International Conference on Software Engineering Advances, pp. 10---10, (2007)

106. Savage, T., Revelle, M., and Poshyvanyk, D., “Flat3: Feature Location and Textual Tracing Tool,” Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, pp. 255-258, (2010)

107. Schaefer, I., Bettini, L., Damiani, F., and Tanzarella, N., “Delta-Oriented Programming of Software Product Lines,” Proceedings of the 14th international conference on Software product lines: going beyond, pp. 77-91, (2010)

108. Schmid, K. and Eichelberger, H.: A Requirements-Based Taxonomy of Software Product Line Evolution. Electronic Communications of the EASST. 8 (2008)

109. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y.: Generic Semantics of Feature Diagrams. Comput. Netw. 51, 456-479 (2007)

110. Schulze, S., Thüm, T., Kuhlemann, M., and Saake, G., "Variant-Preserving Refactoring in Feature-Oriented Software Product Lines," Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, pp. 73-81, (2012)

111. Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P.: Automated Merging of Feature Models Using Graph Transformations. In: Generative and Transformational Techniques in Software Engineering Ii, pp. 489-505. Springer (2008)

112. Seidl, C., Heidenreich, F., and Uwe, A., "Co-Evolution of Models and Feature Mapping in Software Product Lines," Proceedings of the 16th International Software Product Line Conference - Volume 1, pp. 76-85, (2012)

113. She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K., "The Variability Model of the Linux Kernel," VaMoS, pp. 45-51, (2010)

114. She, S., Lotufo, R., Berger, T., Wąsowski, A., and Czarnecki, K., "Reverse Engineering Feature Models," Proceedings of the 33rd International Conference on Software Engineering, pp. 461-470, (2011)

115. Solheim, I. and Stølen, K., "Technology Research Explained, Technical Report A313," (2007)

116. Stone, A. and Sawyer, P.: Identifying Tacit Knowledge-Based Requirements. IEE Proceedings-Software. 153, 211-218 (2006)

117. Sun, J., Zhang, H., Fang, Y., and Wang, L.H., "Formal Semantics and Verification for Feature Modeling," Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pp. 303-312, (2005)

118. Svahnberg, M. and Bosch, J., "A Case Study on Product Line Architecture Evolution," Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99), (1999)

119. Svahnberg, M. and Bosch, J., "Characterizing Evolution in Product Line Architectures," Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications, pp. 92-97, (1999)

120. Svahnberg, M. and Bosch, J.: Evolution in Software Product Lines: Two Cases. Journal of Software Maintenance: Research and Practice. 11, 391-422 (1999)

121. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling," Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, pp. 128-142, (2008)

122. Svendsen, A., Zhang, X., Haugen, O., and Møller-Pedersen, B., "Towards Evolution of Generic Variability Models," Proceedings of the 14th international conference on Models in Software Engineering, pp. 53-67, (2012)

123. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, O., Møller-Pedersen, B., and Olsen, G.K., "Developing a Software Product Line for Train Control: A Case Study of

Cvl," Proceedings of the 14th international conference on Software product lines: going beyond, pp. 106-120, (2010)

124. Saaltink, M.: The Z/Eves System. In: Zum'97: The Z Formal Specification Notation, pp. 72-85. Springer (1997)

125. Thum, T., Kastner, C., Erdweg, S., and Siegmund, N., "Abstract Features in Feature Modeling," Proceedings of the 15th International Software Product Line Conference, pp. 191-200, (2011)

126. Van Deursen, A. and Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology. 10, 1-17 (2002)

127. Van Gurp, J., Bosch, J., and Svahnberg, M., "On the Notion of Variability in Software Product Lines," Proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 45-54, (2001)

128. van Gurp, J. and Savolainen, J., "Service Grid Variability Realization," Proceedings of the 10th International on Software Product Line Conference, pp. 85-94, (2006)

129. Vasilevskiy, A., "Conquering Overlapping Fragments in Cvl", University of Oslo, (2013)

130. Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., and Heider, W., "Flexible and Scalable Consistency Checking on Product Line Variability Models," Proceedings of the IEEE/ACM international conference on Automated software engineering, pp. 63-72, (2010)

131. Weiss, D.M., "Commonality Analysis: A Systematic Process for Defining Families," Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, pp. 214-222, (1998)

132. Weiss, D.M., "Family-Oriented Abstraction Specification and Translation: The Fast Process," Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland, pp. 14-22, (1996)

133. Weiss, D.M. and others: Software Product-Line Engineering: A Family-Based Software Development Process. (1999)

134. Weston, N., Chitchyan, R., and Rashid, A., "A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements," Proceedings of the 13th International Software Product Line Conference, pp. 211-220, (2009)

135. Whyte, W.F.E.: Participatory Action Research. Sage Publications, Inc, (1991)

136. Wile, D., "Lessons Learned from Real Dsl Experiments," Proceedings of the 36th Annual Hawaii International Conference on System Sciences, pp. 10-pp, (2003)

137. Woodcock, J. and Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Inc., (1996)

138. Yin, R.K.: Case Study Research: Design and Methods. Sage, (2009)

139. Zave, P.: An Experiment in Feature Engineering. In: Programming Methodology, pp. 353-377. Springer (2003)

140. Zhang, X., "Synthesize Software Product Line," Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, pp. 341-342, (2010)

141. Zhang, X., Haugen, O., and Moller-Pedersen, B., "Augmenting Product Lines," Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01, pp. 766-771, (2012)

142. Zhang, X., Haugen, O., and Moller-Pedersen, B., "Model Comparison to Synthesize a Model-Driven Software Product Line," Proceedings of the 2011 15th International Software Product Line Conference, pp. 90-99, (2011)

143. Zhang, X., Haugen, Ø., and Møller-Pedersen, B., "Semantic Differencing for Product Line Evolution," SINTEF A25398 ISBN 978-82-14-05332-6 (2013)

144. Zhang, X., Lin, Y., and Haugen, Ø., "April: A Dsl for Payroll Reporting," The 1st International Workshop on Future trends of Model-Driven Development in conjunction with the 11th International Conference on Enterprise Information Systems, Milan, Italy, (2009)

145. Zhang, X. and Møller-Pedersen, B., "Towards Correct Product Derivation in Model-Driven Product Lines," Proceedings of the 7th international conference on System Analysis and Modeling: theory and practice, pp. 179-197, (2013)

146. Ziadi, T., Frias, L., da Silva, M.A.A., and Ziane, M., "Feature Identification from the Source Code of Product Variants," Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pp. 417-422, (2012)

147. Ziadi, T., Hélouët, L., and Jézéquel, J.-M.: Towards a Uml Profile for Software Product Lines. In: Software Product-Family Engineering, pp. 129-139. Springer (2004)

148. Ziadi, T. and Jézéquel, J.-M.: Software Product Line Engineering with the Uml: Deriving Products. In: Software Product Lines, pp. 557-588. Springer (2006)

Appendix I

Paper I: Developing a Software Product Line for Train Control: A Case Study of CVL

Authors: Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg et al.

Journal: Proceedings of the 14th international conference on Software product lines: going beyond, Springer-Verlag, 2010, 106-120.

Author contribution: Xiaorui Zhang is one of the main contributors of this paper, and has contributed to all parts of it (tool implementation, case study design & execution and paper-writing), responsible for 40% of the work.

Appendix II

Paper II: APRiL: A DSL for Payroll Reporting

Authors: Xiaorui Zhang, Yun Lin and Øystein Haugen

Journal: The 1st International Workshop on Future trends of Model-Driven Development in conjunction with the 11th International Conference on Enterprise Information Systems, Milan, Italy, 2009

Author contribution: Xiaorui Zhang is the main contributors of this paper, and has contributed to all parts of it (tool implementation, case study design & execution and paper-writing), responsible for 90% of the work.

Appendix III

Paper III: Model Comparison to Synthesize a Model-Driven Software Product Line

Authors: Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen

Journal: Proceedings of the 15th International Software Product Line Conference, IEEE Computer Society, 2011, 90-99

Author contribution: Xiaorui Zhang is the main contributor of this paper, and has contributed to all parts of it (ideas, tool implementation, paper-writing and all topics of the paper), responsible for 90% of the work.

Appendix IV

Paper IV: Towards Correct Product Derivation in Model-Driven Product Lines

Authors: Xiaorui Zhang and Birger Møller-Pedersen

Journal: Proceedings of the 7th international conference on System Analysis and Modeling: theory and practice, Springer-Verlag, 2013, 179-197.

Author contribution: Xiaorui Zhang is the main contributor of this paper, and has contributed to all parts of it (ideas, tool implementation, paper-writing and all topics of the paper), responsible for 90% of the work.

Appendix V

Paper V: Augmenting Product Lines

Authors: Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen

Journal: Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01, IEEE Computer Society, 2012, 766-771.

Author contribution: Xiaorui Zhang is the main contributor of this paper, and has contributed to all parts of it (ideas, tool implementation, paper-writing and all topics of the paper), responsible for 90% of the work.

Appendix VI

Paper VI: Towards Evolution of Generic Variability Models

Authors: Andreas Svendsen, Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen

Journal: Proceedings of the 14th international conference on Models in Software Engineering, Springer-Verlag, 2012, 53-67.

Author contribution: Xiaorui Zhang is one of the main contributors of this paper, and has contributed to all parts of it (ideas, tool implementation, paper-writing and all topics of the paper), responsible for 45% of the work.

Appendix VII

Paper VII: Semantic Differencing for Product Line Evolution

Authors: Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen

Journal: SINTEF Report A25398, ISBN 978-82-14-05332-6

Author contribution: Xiaorui Zhang is the main contributor of this paper, and has contributed to all parts of it (ideas, tool implementation, paper-writing and all topics of the paper), responsible for 90% of the work.