

Variability-Modelling Practices in Industrial Software Product Lines: A Qualitative Study

by

Divya Karunakaran Nair

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013
© Divya Karunakaran Nair 2013

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Many organizations have transitioned from single-systems development to product-line development with the goal of increasing productivity and facilitating mass customization. Variability modelling is a key activity in software product-line development that deals with the explicit representation of variability using dedicated models. Variability models specify points of variability and their variants in a product line. Although many variability-modelling notations and tools have been designed by researchers and practitioners, very little is known about their usage, actual benefits or challenges. Existing studies mostly describe product-line practices in general, with little focus on variability modelling. We address this gap through a qualitative study on variability-modelling practices in medium- and large-scale companies using two empirical methods: surveys and interviews. We investigated companies' variability-modelling practices and experiences with the aim to gather information on 1) the methods and strategies used to create and manage variability models, 2) the tools and notations used for variability modelling, 3) the perceived values and challenges of variability modelling, and 4) the core characteristics of their variability models. Our results show that variability models are often created by re-engineering existing products into a product line. All of the interviewees and the majority of survey participants indicated that they represent variability using separate variability models rather than annotative approaches. We found that developers use variability models for many purposes, such as the visualization of variabilities, configuration of products, and scoping of products. Although we observed that high degree of heterogeneity exists in the variability-modelling notations and tools used by organizations, feature-based notations and tools are the most common. We saw huge differences in the sizes of variability models and their contents, which indicate that variability models can have different use cases depending on the organization. Most of our study participants reported complexity challenges that were related mainly to the visualization and evolution of variability models, and dependency management. In addition, reports from interviews suggest that product-line adoption and variability modelling have forced developers to think in terms of a product-line scenario rather than a product-based scenario.

Acknowledgements

First, I would like to thank my supervisor, Professor Joanne M. Atlee for providing the necessary guidance in completing my Master's degree. Her assistance has helped me in choosing my research topic, in improving my language skills, and in guiding me with the process of research. I am also grateful to her for granting an internship opportunity at General Motors to explore more about my research topic and to proceed with my research. This thesis would not have been possible without her guidance and active involvement.

Next, I would like to express my sincere gratitude towards Professor Krzysztof and Professor Reid Holmes for devoting their time to read and provide necessary corrections to improve this thesis. Professor Krzysztof has been kind enough to provide professional advice and in introducing me to his research group, which opened a door for acquiring extensive knowledge on my research topic, by collaborating with his group members, especially Thorsten Berger and Professor Andrzej Wasowski of the IT University of Copenhagen. My frequent discussions with Professor Krzysztof's research group have assisted me in validating my course of research.

My colleagues in the Formal Methods Lab have presented me with a wonderful environment to work and in understanding more about the research process. I am grateful to them in cheering me up in occasions when I badly needed breaks from my busy schedule.

I am greatly indebted to my husband, Sivan, for providing motivation and great mental strength, and in taking time to patiently listen to my occasional issues, during the course of this Masters research. He has always reminded me to keep an open mind and has assured me of positive results. In addition, he has dedicated a good portion of his time in proof-reading my thesis and in suggesting necessary editions. He cared deeply for me when I was passing through stressful times and could not care about myself.

Next, I would like to express my gratitude towards my family for their unconditional love and support. It was with their encouragement that I was able to choose and excel in the field of Computer Science. Finally, I would like to thank the Great Almighty for granting me this opportunity and in providing me with great strength, confidence, and positive attitude to fulfill my thesis.

Contents

List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Problem Statement.....	2
1.2 Thesis Organization	3
2 Background and Related Work	4
2.1 SPL Terminology.....	4
2.2 Product Line Development Processes and Adoption Strategies.....	5
2.2.1 SPL Adoption Strategies.....	6
2.3 Variability Modelling Notations and Tools.....	6
2.3.1 Feature-based Variability Modelling.....	7
2.3.2 Variability Modelling Tools.....	9
2.4 Product-Configuration Approaches.....	10
2.5 Empirical Work on Variability Modelling and Software Product Lines.....	10
3 Qualitative Study: Part 1- Survey and Results	14
3.1 Survey Design and Questionnaire Distribution.....	14
3.2 Survey Results.....	15
3.2.1 Background of Participants.....	16
3.2.2 Contextual Information on Variability Modelling.....	17
3.2.2.1 Application Domains.....	17
3.2.2.2 Product-line Adoption.....	18
3.2.2.3 Product-line Artifacts.....	19
3.2.3 Perceived Value on Variability Modelling.....	20
3.2.4 Notations and Tools for Variability Modelling.....	21
3.2.5 Scales and Details of Variability Models.....	25
3.2.6 Complexity Challenges and Mitigation Strategies.....	27

3.2.7	Additional Comments by Participants.....	29
3.3	Threats to Validity.....	30
4	Qualitative Study: Part 2- Interviews and Results.....	31
4.1	Interview Design and Participant Sources.....	31
4.2	Interview Results.....	33
4.2.1	Case Study 1: Global Producer of Electronic and Mechanical Components.....	33
4.2.2	Case Study 2: Consulting Company for Web-Based Applications.....	38
4.2.3	Case Study 3: IT Consulting Company for Enterprise, Embedded and Mobile Applications.....	43
4.2.4	Case Study 4: Automotive Company.....	46
4.2.5	Case Study 5: BigLever GEARS.....	51
4.2.6	Case Study 6: Pure::Variants from Pure Systems.....	58
4.2.6.1	Interview 1.....	59
4.2.6.2	Interview 2.....	62
4.2.6.3	Comparison of Results.....	66
4.3	Discussion of Interview Results and Findings.....	68
4.4	Threats to Validity.....	79
5	Conclusions	80
	Bibliography	82
	Appendix A	94
	Appendix B	95

List of Tables

3.2.2	Table 1: Distribution of participants' domains.....	18
3.2.4	Table 2: Usage of home-grown, commercial, and open-source variability tools.....	24
3.2.5	Table 3: Distribution of responses on sizes of variability models.....	26
4.1	Table 4: Summary of interviewees in case studies.....	33
4.2.6	Table 5: Comparison of interview 1 and interview 2.....	68
4.3	Table 6: Details on variability models: tools used, type of product-line artifacts, and variability realization mechanisms.....	75

List of Figures

2.3.1	Figure 1: Feature model for a mobile phone system.....	8
3.2	Figure 2: Geo-data showing the origin of participants.....	16
3.2.1	Figure 3: Pie chart showing respondents' experience with SPLE.....	16
3.2.1	Figure 4: Roles of participants.....	17
3.2.2.2	Figure 5: Response distribution on product-line adoption strategies.....	18
3.2.2.3	Figure 6: Response distribution on artifacts used to model variabilities.....	19
3.2.3	Figure 7: Response distribution on respondents' attitude towards variability modelling.....	20
3.2.3	Figure 8: Response distribution on uses of variability modelling.....	21
3.2.4	Figure 9: Response distribution for type of variability representation.....	22
3.2.4	Figure 10: Response distribution on variability modelling notations.....	22
3.2.4	Figure 11: Response distribution on variability modelling tools.....	23
3.2.5	Figure 12: Response distribution on types of variability units.....	25
3.2.5	Figure 14: Response distribution on the use of cross-tree constraints.....	27
3.2.6	Figure 15: Response distribution on complexity areas in variability modelling.....	28
3.2.6	Figure 16: Response distribution on mitigation strategies.....	29

Chapter 1

Introduction

Modern software systems possess an increasing amount of variability. Variability allows a software system to be useful in a variety of contexts, increasing reusability. Consider the range of product variations offered by a typical car manufacturer: a series of brands (e.g., Chevrolet, Cadillac, GMC), where each brand is further classified into different vehicle types according to usage, such as passenger vehicles, commercial vehicles, and special vehicles (e.g., ambulances, police cars). Each vehicle type can be further decomposed into vehicle models (e.g., Sedan, Coupe), with varying styles of their own (e.g., Coupe 2-seater, Coupe 4-seater). Each of these vehicle models has its own configuration of *mandatory* and *optional* features, while sharing a few common features with other models of the same vehicle type. To increase product quality and reduce cost and time-to-market, it is in the best interest of the automotive manufacturer to reuse existing product artifacts to build vehicle variants in a systematic way, using software product-line development.

A software product line (SPL) consists of a collection of software assets (features) that are used in the construction of a family of related software systems. An SPL is structured such that each product can be systematically developed from the SPL's set of assets [1] according to the needs of a specific market segment. The core assets in an SPL have specific locations where variability can be introduced. To derive a customized product, these locations of variability are bound to a chosen variation option, thereby selecting the SPL assets to be incorporated into the product. A new product is added to the product line by reusing existing core assets and, in some cases, by incorporating new modules or components that satisfy the specific functionalities of the new product. According to Stahl, Voelter, and Czarnecki [2], the efficiency of an SPL approach is based on the degree to which variability is managed, from the initial stages of SPL development to the final stages of product derivation.

An important activity in product-line development is variability modelling. Variability modelling has received tremendous attention in the past few decades and is the basis for both research-based [1, 3, 4] and industrial product-line methodologies [5, 6]. Variability modelling deals with the explicit representation of variability using dedicated models that specify common and variable features that make up products in a software product line. Modelling variability is beneficial for many product-line activities, including product configuration, domain modelling, planning, and marketing. With the growth of modern complex software systems, the challenges for modelling and managing variability have

increased; some of these challenges include representing large numbers of variable requirements for different products and releases, modelling and managing variations and their dependencies among different SPL artifacts and assets, and configuring and optimizing software products. In addition, we must acknowledge the different types of variabilities that exist in modern software systems, including functional variability (variations in functional behaviours), non functional variability (variations in non-functional system properties), fault-based variability (variations with respect to different faults), and so on. Managing these diverse variabilities is extremely complex and requires sophisticated modelling techniques. Furthermore, as new variabilities are added or existing variabilities are updated, we need to propagate the changes to existing artifacts and keep them consistent.

Over the past few decades, many organizations have transitioned from single-systems development to software product lines, in order to increase software customization, shorten the time-to-market, and improve the quality of products. However, there are many challenges that organizations face with respect to adopting a product-line approach and in modelling and managing the variations in their product lines, such as the need for systematic methods to identify and exploit variabilities, appropriate visualization and modelling methods that scale to tens of thousands of variations, efficient methods to detect and resolve dependency interactions among variations at different abstraction levels, and techniques to trace variability information among development artifacts [7].

1.1 Problem Statement and Contributions

Although many variability-modelling notations and tools have been designed by researchers and practitioners, very little is known about their usage, actual benefits and challenges. In the past few decades, experimental and exploratory research (e.g., experience reports, case studies) has been conducted to study the industrial applications of product lines. However, the majority of these reports describe product-line practices in general, with little focus on variability modelling. There are a few empirical reports that try to address variability modelling in industrial contexts; however, most of them do not discuss specific details of variability models (such as their sizes or types of dependencies), or are not based on hard empirical data. This is a serious concern, especially given the crucial role that variability modelling plays in software product-line engineering (SPLE). Recent literature studies [8, 9, 10, 11] emphasize the large number of variability-management and modelling approaches proposed, but their applicability in practice and thorough evaluation is a major gap in SPLE research. This lack of knowledge about contemporary practices threatens the development and improvement of variability-modelling approaches. Furthermore, a detailed understanding of the practical applications of variability models is important to improve the existing notations and tools.

This thesis aims to address the research gap described above through an empirical study on the industrial usage of variability modelling, and its benefits and challenges. Our qualitative study uses two empirical

methods: (1) a questionnaire-based survey and (2) semi-structured interviews. We report the variability-modelling practices in a set of medium- and large-scale companies that apply SPLE and variability modelling, from the perspective of the study participants who are employees of the companies. The questionnaire-based survey helps to obtain a preliminary record of industrial practices and to identify participants for interviews. On the other hand, interviews provide a clear and detailed qualitative record of actual practices and of proprietary variability models.

Our work is exploratory, guided by the following research questions:

RQ1: What are the different methodologies and tools applied to model variability?

We gathered information about companies' conventions in modelling variability, such as their strategies to (1) identify units of variations, (2) build variability models, (3) modularize variability models, (4) map variability models to other software-development artifacts, and (5) evolve models. In addition, we asked about the different variability-modelling notations and tools used by the organizations.

RQ2: What are the perceived values and challenges of variability modelling?

We asked about the benefits of variability modelling (beyond its use to configure products). We also elicited the challenges that practitioners face when applying common or in-house variability-modelling techniques, and asked about recommendations that the participants might have to improve their existing variability-modelling processes or product-line environment.

RQ3: What are the contents, structures, and sizes of variability models?

We gathered information on the units of variations, the sizes of variability models, and the contents of models, and compared these characteristics with those of published academic variability models.

1.2 Thesis Organization

We proceed as follows. In chapter 2, we provide an adequate background for understanding this thesis, including definitions and a brief overview of SPLE processes and variability-modelling methods and tools. We also present related empirical studies of variability modelling. Chapter 3 describes the survey design and presents the survey results and our findings. In Chapter 4, we describe the interview design, participant selection criteria, and the results from seven interviews. Finally, Chapter 5 concludes this thesis and proposes possible future work.

Chapter 2

Background and Related Work

This chapter presents (a) the necessary background for understanding this thesis, including an overview of SPLE processes and variability-modelling methods and tools; and (b) an overview of existing approaches and empirical work on variability modelling.

2.1 SPL Terminology

The research literature provides a rich set of terms used in SPL development, but there are inconsistencies in the usage of these terms [12]. Hence, in this sub-section, we provide a set of SPL-related terms and definitions that are borrowed from the SPL literature and that we will use throughout the rest of this document.

Software Product Lines and Software Product-Line Engineering:

“A software product line is a set of software-intensive systems that share a common feature set and a platform satisfying a particular market segment’s specific needs or mission, and that are developed from a common set of core assets in a prescribed way” [13]. A common platform refers to a basic set of technologies on which multiple systems are built. The main reasons for adopting a software product-line approach in an organization are to reduce development costs, reduce time-to-market, improve quality, and decrease maintenance effort [14]. “Software Product-Line Engineering (SPLE) is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization” [14]. Mass customization refers to the large-scale production and specialization of software products according to the needs of different classes of customers. SPLE aims at systematic reuse of software.

Variability, Variant, and Variation Points:

“Variability or variation is an assumption about how each product in a product line differs from each other” [15]; that is, variability supports the ability of a system to be customized or changed according to the varying expectations of different classes of customers. Variability exists at different levels of abstraction in SPL development, including requirements variability (mainly feature based), architecture variability (mainly component based), and implementation variability (mainly code based). Variability is realized using two concepts: variant and variation points. A variation point is defined as a specific

location in an SPL artifact where variability occurs, and a variant is a single choice for a variation point. Even though variants and variation points are associated with SPL artifacts, they are considered to be self-contained entities, separate from the SPL artifacts [14].

Software Product-Line Asset and Artifact:

“A software product-line asset (also known as core asset) is an artifact or a set of artifacts that can be reused for developing multiple software products” [13]. An artifact is any piece of hardware, software, or documentation that helps with the development of a software system. According to Klaus Pohl [14], there are two kinds of artifacts in product-line development: domain artifacts and application artifacts. Domain artifacts are reusable artifacts created during product-line development, and application artifacts are artifacts developed for specific applications within the product line.

Product Configuration (or Product Profile):

“A product configuration is a specific set of variants selected for a particular software product, in order to customize it for a particular customer or a market segment” [16]. Product configuration binds variations to customer-specific choices; the result is an instance of the product line, otherwise called a product instance or a configured product.

Binding Time:

“Binding time refers to the latest possible time in SPL development when a variation point should be bound by a variant” [17]. The possible values for binding times are design time, code-generation time, pre-compile time, link time, post-build time, and runtime.

2.2 Software Product Line Development Processes and Adoption Strategies

Most established SPL methodologies consist of two main processes: domain engineering and application engineering [2, 18, 19]. While domain-engineering processes define the methods for achieving variability and for developing reusable artifacts, application-engineering processes realize and bind the selected variabilities to derive products according to the specific needs of customers.

Domain engineering is the process of describing, modelling, and analyzing variabilities and commonalities in a software product-line system. It defines the scope of the family of products to be supported by a product line. The output is a reusable platform (or framework) which comprises all of the different SPL artifacts in a product family, and it guarantees that the defined variations in the SPL artifacts are capable of deriving all of the products in the product family [14, 21].

Application engineering deals with actual product development. It involves deriving SPL applications using the platform and reusable artifacts created during domain engineering [14]. In application engineering, a product configuration is derived by carefully selecting and realizing the variations in domain-engineering artifacts the use of specific variability-realization mechanisms [77, 78].

2.2.3 SPL Adoption Strategies

There are three SPL adoption strategies defined in the SPL literature by Kruger [21]. All of these strategies have a direct influence on variability modelling.

Proactive: “Product line was developed before any product was derived”. Thus, a variability model is built upfront based on a scoping process.

Reactive: “A single product is evolved into a product line”. An initial variability model covering one product is created by identifying features of the product. This model is then updated by gradually adding more features, which likely requires subsequent refactorings.

Extractive: “Existing similar products are re-engineered into a product line”. This strategy requires analysis of the commonalities and variabilities among the products, and identifying features by abstracting the differences.

The product-line community maintains a Hall of Fame¹ of documented successful SPLE adoptions. To become a member of the Hall of Fame, a product line must be developed according to one of the above-mentioned reference processes. Ideally, it is proactively conceived as a product line from the very beginning. However, the proactive approach imposes greater risks, due to a high initial investment, than the reactive or extractive approaches [21].

2.3 Variability Modelling Notations and Tools

Variability modelling is the key activity to manage variability in product lines. Variability can be represented as annotations to existing artifacts or as separate models. While the former is often applied using annotation facilities of component frameworks, such as Spring [22], the latter represents the most common approach to variability modelling, with many languages and notations available. Among the most popular notations are feature models [23, 24], which describe the common and variable characteristics of the product line in terms of a hierarchy of features. In fact, a recent literature review determined that 33 of 91 approaches to variability management are based on feature models [25]. Other popular techniques include decision modelling [26, 27], goal modelling [28, 29], UML-based approaches [30, 31], DSLs [32], ADL-based approaches [60, 61, 62, 63, 64, 65], and languages that model variation points [33, 34, 7]. The majority of participants in our qualitative studies (survey, interviews) use feature modelling, and hence we only describe feature-modelling notations and feature-based tools.

¹ <http://www.splc.net/fame.html>

2.3.1 Feature-based Variability Modelling

A feature model depicts a set of configuration choices, in terms of feature-selection decisions, that have to be made when configuring products. A feature is any observable characteristic of a system that is useful to its stakeholders [23]. One of the advantages of using feature-based concepts is that features are understandable by stakeholders from diverse backgrounds [35].

Feature-Oriented Domain Analysis (FODA) [23] was one of the first approaches to systematically manage variability using feature concepts. The main intention of FODA is to capture commonalities and variabilities of a product-line system during early development stages. A feature diagram (FD) is a tree-based notation that models features in a hierarchy. The root of the hierarchy represents a main concept in the system under development (SUD), and is decomposed into more fine-grained features (sub-features). Each feature in a FD is associated with a set of requirements. A FD relates features to each other through dependencies and constraints. Depending on the type of feature decomposition, a feature can be *mandatory*, *optional*, or an *alternative* to another feature or group of features. The optional features and the feature groups specify variability within a feature model. In addition to feature dependencies, cross-tree constraints apply to features that are not related by decomposition (e.g., *requires*, *excludes*). For instance, if a feature *X* is selected, and if there exists a relation '*X requires Y*', then feature *Y* must be selected as well.

Figure 1 shows a small sample feature model for a mobile phone, adapted from [26, 85]. The root *Mobile Phone* is decomposed into four sub-features: *CallProcesing*, *Multimedia*, *Resolution*, and *EarPhone*. *CallProcesing* and *Resolution* are *mandatory* (solid dot), while the other two are *optional* (hollow dot). The children of *Multimedia* and *Resolution* each participate in feature groups: *OR* groups (filled arc) require that at least one member feature be selected, whereas *XOR* groups (hollow arc) require that exactly one feature member be selected. Additional cross-tree constraints further restrict possible combinations of mobile-phone features: *MP3* support requires an *EarPhone*, and a low *Resolution* of *240x400* excludes a *Camera*. Each feature can be a Boolean, string, or an integer type. For example, *EarPhone* is a Boolean feature indicating the presence or absence of an ear phone in a mobile phone; *NumCalls* is of type integer which represents the number of calls; and *Custom* is of type string allowing customized values for the *Resolution* of the mobile phone.

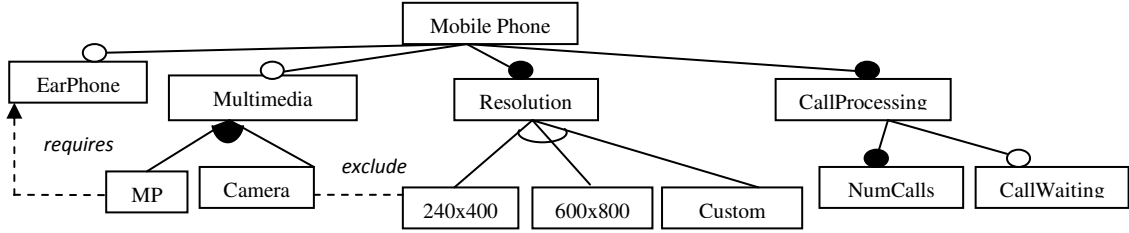


Figure 1: Feature model for a mobile phone system, adapted from [26, 85]

Many researchers propose extensions to FODA to improve expressiveness. FeatuRSEB [36] supports use-case-based feature modelling by combining FODA and reuse-driven software-engineering business (RSEB) methods. FeatuRSEB represents variability by extending the structure of use-case and object diagrams with variants and variation points. Jan Bosch and Jilles van Gurp extended FeatuRSEB by associating binding times to features, using additional feature constructs in the feature diagram. FORM [37] and FOPLE [38] extend FODA to include software-design aspects. These approaches use four layers to model different views on software development according to the level of abstraction: a capability layer, a domain-technology layer, an operating-environment layer, and an implementation layer. However, this decomposition compromises the simplicity and comprehensibility of FODA models, and further increases the complexity of feature modelling. Moreover, neither of these two approaches model feature dependencies that exist among each layer. Inspired by FORM, Brown et al. [39] present a bi-directional feature modelling strategy to capture commonalities and variabilities in product-line systems, by associating software features to hardware features. This approach uses feature relations, such as *provided-by* and *consists-of*, which are borrowed from existing feature-modelling approaches [37, 40]. Riebisch et al. [41, 42] analyzed existing feature-based notations and found that “the combinations of *mandatory* and *optional* features with *alternatives*, *OR*, and *XOR* relations could lead to ambiguities”. To reduce ambiguity, Riebisch et al. proposed a new feature-based notation that focuses on expressing multiplicities among groups of features in a feature diagram. Ye et al. [43] proposed two views: (1) a feature-tree view to represent hierarchical dependencies among features, and (2) a dependency view to represent non-hierarchical (constraint) relations among features. The feature dependency and constraint relations used in this approach are similar to FODA feature relations. The authors proved that their approach is adequate to model evolving product lines.

Inspired by the above approaches, Czarnecki et al. [44] introduced the notion of cardinality-based feature modelling (CBFM) in which a variation point can be annotated with cardinality constraints on feature selection. For a feature with a set of sub-features, cardinality is usually indicated by a $\langle m, n \rangle$ interval where m represents the minimum number of sub-features to be selected and n represents the maximum number of sub-features to be selected for a feature configuration. In addition, Czarnecki [45] proposes a template-based approach to map feature models to requirements models, such as activity and class

diagrams. The approach represents a product-line family by a feature model and a model template. A model template is a union of model elements present in all valid instances of a given requirement model, such as an activity diagram. A presence condition can be attached to a model element, which indicates whether the model element is present or not in a particular instance of the requirement model. These presence conditions are usually Boolean conditions, which are defined in terms of the features or feature attributes in the feature model. The approach performs automatic instantiation of the model template with respect to a given feature configuration; the result is a model template instance augmented with presence conditions. Rieser and Weber [46] introduce multi-level feature trees for large-scale industrial product lines to manage different feature trees that correspond to a hierarchy of product lines supported by an organization. This work is inspired by Czarnecki [45]; it uses a single reference feature model which acts as a template, from which various feature trees are derived for different product lines.

Lee and Kang [47] propose a feature-oriented approach for managing variation points for dynamically reconfigurable core assets. This approach concentrates on determining variant reconfiguration contexts and strategies. It refines feature models by analyzing feature bindings and grouping features into binding units based on their binding times. This feature-binding analysis can identify binding dependencies among variation points at run time.

The Common Variability Language (CVL) [48, 49] is an emerging standard for modelling and managing variability in SPLs. Variability modelling in CVL is based on feature-based concepts. CVL incorporates variability into existing product-line-model artifacts without modifying them. The CVL architecture consists of three layers. The first layer consists of models that conform to the meta-object facility (MOF). The second layer consists of two parts: (1) *variability realization*, which introduces variability through variation points in the base model, and (2) *variability abstraction*, which specifies variabilities in terms of variability specifications called *VSpecs*. A *VSpec* is similar to a feature-model: variability is expressed using binary-decision choices, value assignments on attributes, and substitution of model fragments. The final layer is the variability encapsulation layer, which modularizes variability. The CVL approach can be applied to all MOF-based models independent of the level of abstraction.

2.3.2 Variability Modelling Tools

GEARS [50] from BigLever and pure::variants [51] from Pure Systems are two commercial tools that are commonly used to model variability in industrial SPLs. Both tools support a textual and graphical representation for feature diagrams, and they model dependencies and constraints among features. Unlike GEARS, pure::variants provides additional support for expressing complex constraint relations, in Prolog. A family model is defined in pure::variants for mapping artifacts (especially components) to features, whereas the concept of a variation point is used for mapping in GEARS.

There also exist a number of freely available tools for feature-based variability modelling and configuration, such as FeatureIDE [52], AHEAD [53], and FMP [54]. CONSUL [55] is a freely available tool chain developed for SPLs; CONSUL@GUI uses an improved version of FODA feature diagrams to represent variability in the problem space and to visualize feature configurations. Software Product Lines Online Tools (SPLOT) [56] is a similar tool chain that includes web-based interactive tools to represent, analyze, and configure feature models. SPLOT maintains a repository of automatically generated feature models with the aim to support qualitative studies involving analysis and reasoning on feature models. Some companies use specific home-grown tools for modelling and configuring variability; in a few cases, a modelling language with variability extensions is used (e.g., variability stereotypes or constructs in UML and Simulink). A significant percentage of our survey participants use home-grown tools, domain-specific tools and open-source tools to model variability in their product-line projects.

2.4 Product-Configuration Approaches

As mentioned earlier, product configuration is performed during application engineering, and the output is a product that satisfies the requirements of a specific customer or a market segment [73]. The product-configuration activity involves the process of (a) selecting features or decisions based on the interests of a customer; and (b) deriving an executable product based on the decision-choices, using product-generation tools or configurators. In order to achieve better results during product configuration, constraints are introduced to protect against invalid choices and derive optimal configurations that satisfy user's functional and non-functional needs.

There exist configuration languages and tools (e.g., XVCL [74]) to model and guide the derivation of product configurations. In addition, various variability-realization mechanisms (e.g., template instantiation [77], parameterization [77, 78], configuration files [77, 75, 78], conditional compilation [77, 78], and design patterns [76]) are described in the SPL literature to configure and realize variability in product-line artifacts and to derive products. Highly-referenced approaches to product configuration include the use of configuration models and languages [74, 90, 91], and the use of product configurators [65, 82, 84] to automatically derive product configurations.

2.5 Empirical Work on Variability Modelling and Software Product Lines

Multiple surveys, case studies, and literature reviews have been conducted to understand the efforts of product-line development in industry. The majorities of these works cover product-line processes in general and do not focus on the practices related to tools, notations, and methods used for modelling variability. In contrast, this thesis reports on current industrial practices specific to variability modelling and on the type of product-line adoption strategy used by organizations.

Mohagheghi and Conradi [92] present a review of systematic software reuse in industrial settings, based on experience reports and quantitative data from a selected set of journal and conference papers published between 1994 and 2005. This report provides empirical evidence on how efficient reuse can positively influence software quality and productivity. Along similar lines, Verlage and Kiesgen [93], Thorn [94], Thorn and Gustafsson [95], and John et al. [96] report the advantages of using a product-line approach in an organization; they compare various product-line practices, applied to a few selected small- and medium-sized enterprises (SMEs). These works are similar to ours with respect to identifying various variability-modelling approaches ranging from ad-hoc to sophisticated approaches. However, their focus is mainly towards establishing a correlation between successful organizational practices and the size of organization. Bergey et al. [97], van der Linden et al. [98], and Birk [3] describe industrial product-line practices in large-scale organizations by citing detailed examples of existing case studies from the literature; however, these reports focus mainly on practices concerning organizational design, team collaboration, and project management, with very little detail on variability modelling. The Software Engineering Institute has published a catalog of case studies [99] that discuss successful product-line adoption in both small- and large-scale organizations. None the above-mentioned works cover the structure and content of variability models, such as types of features and their relations, as well as the practices, methods, process and tools used for variability modelling. Our work complements these existing works to include variability-modelling practices that span international medium- and large-scale organizations.

Similar to the above-mentioned reports, case studies, and literature reviews, there exist many experience reports in the product-line community's Hall of Fame¹ that discuss successful industrial adoptions of product lines. Although all these reports provide detailed accounts of organizational, economic, and process-based viewpoints of specific product-line projects, very few of the reports describe practices that concern variability modelling and with only a limited amount of detail. On the other hand, reports by Grunbacher et al. [100], Riebisch et al. [42], Reiser et al. [101], and Gillan et al. [102] present relatively detailed insight into variability-modelling practices within specific product-line organizations. Grunbacher et al. report on industrial experiences in building and managing variability models using two customized Eclipse-based case tool suites (DOPLER and an industry-specific maintenance tool) that satisfy different organizational needs. The authors conclude that it is better for companies to use different tools optimized for specific purposes rather than relying on a one-size-fits-all solution. They recommend developing software tools themselves as product lines. While the focus of their work is mainly confined to how tools can be efficiently adapted and customized to support variability in industrial product lines, our work provides a broader perspective on industrial variability-modelling practices. Riebisch et al.'s investigate the industrial applicability of feature-based variability-modelling methods and tools in industrial object-oriented product lines using discussions and contributions from the ECOOP'03 workshop. The workshop discussion presented two categories of open questions: (a) definition and usage

of feature models, and (b) management of feature models. Several variability-modelling issues are identified as part of the discussion, including the need for adequate methods to manage complexity. Their report includes most of the variability-modelling aspects covered in our study. In contrast to our study, the specific details on industrial variability models (e.g. size, structure) are very limited or are unavailable, since their findings are mainly based on a general discussion. Rieser et al. propose a feature-modelling framework that addresses the challenge of managing heterogeneous development methods used in large-scale industrial system families (a product line consisting of multiple smaller product lines). The framework consolidates several feature-modelling concepts into a unified feature-modelling framework, with the aim to reduce heterogeneity in large-scale product lines. Using the framework, the authors found several requirements that are essential for feature modelling in large-scale industrial system families, including the need to have flexible feature models that can be customized according to organizational needs. Unlike our study, their work does not present specific industrial experiences or practices regarding industrial variability models, and only focuses on the automotive domain. Similar to Rieser's work, Gillan et al. focuses on challenges in adopting feature modelling, but in the telecommunications domain. Some of the challenges include managing large numbers of variation points and the need to express behaviour variations. As a solution, the authors propose an experimental feature-modelling notation and describe how their notation addresses the identified challenges. Similar to Rieser's work, details on industrial variability models are unavailable and their work is limited to one domain, contrary to our study which spans different domains and involves varying organization sizes.

Although all of the above-mentioned reports focus on industrial variability modelling, none of them disclose specific practices or details about industrial variability models and artifacts, such as the sizes of models, the number of dependencies, the types of artifacts involved, and so on.

There are many empirical reports that are published mainly by researchers on product-line and variability- modelling practices, especially in the automotive and telecommunications domains [103, 104, 105, 106, 107, 89, 88]. However, it is not confirmed whether the methods and practices described in these research reports are the industrial participants' actual SPL practices or are exploratory pilot projects or case studies. The empirical work by Chen et al. [10] uses a focus group of eleven industrial practitioners to identify common product-line challenges. Some of the challenges that we identified in our industrial survey confirm their challenges reported in their study. For instance, their study reports the need for a systematic means to identify and express commonalities and variabilities using modelling notations with better visualization capabilities; this challenge was also reported by 59% of our survey participants. In addition, their study reports challenges in evolving feature models, especially in managing dependencies during feature additions, removals or refactorings; this challenge was raised by 56% of our survey participants and by two of our interviewees. A literature review by Hubaux et al. [8] and the work by Chen et al. [9, 11] provide detailed insights into the practical use of variability modelling in industry, and emphasize the need for adequate empirical SPL research on commercial

variability-modelling practices; both of these works motivated us and help to establish the relevance of our qualitative study.

Hubaux et al. [87] conducted a questionnaire-based survey on configuration challenges in open-source systems, such as Linux and eCos systems. Their work emphasizes the need to provide proper assistance to developers in making good configuration choices. Members from our research group [86] extended Hubaux's work and present empirical evidence showing the practical uses of feature-based variability modelling, the size and structure of variability models (e.g., depth), and the types of variabilities modelled for large-scale open-source systems. One of their conclusions relates the industrial application of variability modelling to the growth of commercial tools, such as `pure::variants`. Our study can be considered complementary to their work, in that we cover practices in different domains and organizations, and additionally report on process-based practices and product-line adoption strategies.

Chapter 3

Qualitative Study: Part 1- Survey and Results

This chapter presents our survey of variability-modelling practices in industrial software product lines, which constitutes the first part of our qualitative study on variability-modelling. We first describe the design of the survey questionnaire and the survey-distribution criteria, followed by the survey results and additional comments from participants. We present the questions and their multiple-option answers, exactly as they appear in the questionnaire; the options for answers were designed based on findings from existing SPL literature reports and publications. Please note that this chapter is an extended version of our workshop paper [20] published earlier.

3.1 Survey Design and Questionnaire Distribution

We designed a simple and short questionnaire to be completed by practitioners who apply variability modelling in their product-line projects or who have played a major role in publishing industrial reports on SPLE. Our design of the questionnaire focuses on three topics: (a) variability-modelling notations and tools, (b) the sizes and dimensions of industrial variability models, and (3) the perceived benefits and challenges of product-line adoption and variability modelling.

The questionnaire consists of 15 questions. The first few questions focus on the respondent's opinions towards variability modelling and its usefulness in a product-line environment. The next set of questions focuses on the respondents' industrial experiences, including the notations, tools, and techniques used for variability modelling and realization; details about real-world variability models (such as variability units and scales of variability models) and complexity issues or challenges experienced during variability modelling and the respective mitigation strategies used, if any. The final few questions ask about the context of the respondent's working environment and background, including personal information, experiences with application domains, experience with variability modelling, and the product-line adoption strategy used in their product-line projects. This last set of questions was designed to help us (1) analyze and verify the survey results, (2) classify the domains on which respondents work, and (3) contact the respondent for clarifications or a follow-up interview. Most questions have preset multiple-option answers, plus an open-text region for providing additional information, if desired.

We used the online tool SurveyGizmo² tool to execute the questionnaire. We distributed the questionnaire to our fellow colleagues and researchers who have industrial SPLE experience, our industrial research partners, partners from the Fraunhofer Institute for Experimental Software Engineering (IESE), and employees from organizations listed in the software-product-line Hall of Fame¹. In addition, we distributed the questionnaire at the 2012 International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12) because we considered VaMoS to be an excellent venue for meeting more practitioners and researchers interested in variability modelling. The questionnaire included an invitation for participants to forward the questionnaire to fellow researchers and colleagues, who might be willing to participate in the survey. In the end, we distributed the questionnaire to more than 60 potential participants.

Prior to analyzing the survey results, we filtered and removed the responses from those respondents who claimed on the background questions to have played only the role of a researcher in software product-line projects. This filtering helped us to limit the pool of respondents to industrial practitioners with experience in variability modelling, thus helping to ensure the quality of the results. Analysis of the responses comprised calculating percentage aggregations of participants' responses associated with each individual question and presenting the results in diagrams (e.g., pie chart, bar chart). For each question, we have marked the option(s) chosen by the majority of respondents using a red box. We could not derive any statistical conclusions or correlations because our data-set is limited to a total of 42 responses. Although the multiple-option format restricted us from identifying or measuring any correlations, we were able to obtain a few interesting conjectures as part of our analysis. In the future, researchers can include open-ended questions, which would help to explore more about the responses and obtain meaningful correlations.

3.2 Survey Results

We received a total of 42 responses from participants from 16 different countries: Germany (~24%), Canada (~12%), USA (~12%), Sweden (~7%), Austria (~5%), Norway (~5%), and Spain (~5%). We also received a response from a single participant from each of the following countries: China, Denmark, France, Greece, India, Poland, Switzerland, and the United Kingdom. Figure 2 shows the origins of participants. After filtering out 5 responses from pure researchers and 2 unidentified respondents, who disclosed no personal information, we analyzed the remaining 35 responses; this filtered set of responses included responses from a few researchers or former researchers who have industrial experience with variability modelling and product lining.

¹ <http://splc.net/fame.html>

² <http://www.surveygizmo.com/>

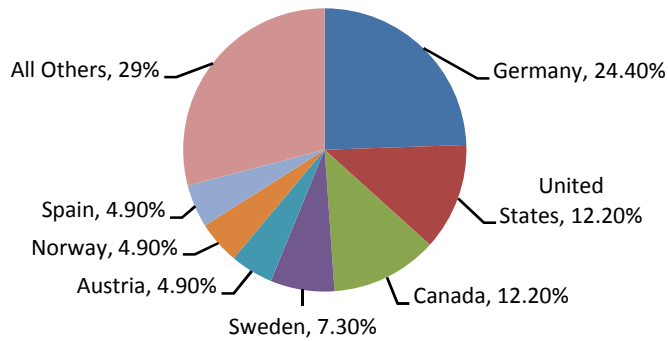


Figure 2: Geo-data showing the origin of participants

3.2.1 Background of Participants

We asked two questions regarding respondents' industrial experience with variability modelling and their respective role(s) in their product-line projects:

"15. How many years of industrial experience do you have in software product line development?"

a) <1 year, b) 1-2 years, c) 3-5 years, **d) 5-10 years**, **e) >10 years**

"14. What have been your roles in product line projects?"

a) Developer, **b) Modeller**, c) Team leader, d) Project Manager, e) Domain expert, f) Product manager, g) Marketing expert h) Researcher

Figure 3 shows the participants' years of experience with SPLE. 56% of the participants indicated having more than 5 years of experience, among which 50% reported having more than 10 years of extensive experience. Very few participants (~8%) had less than one year of professional SPLE experience.

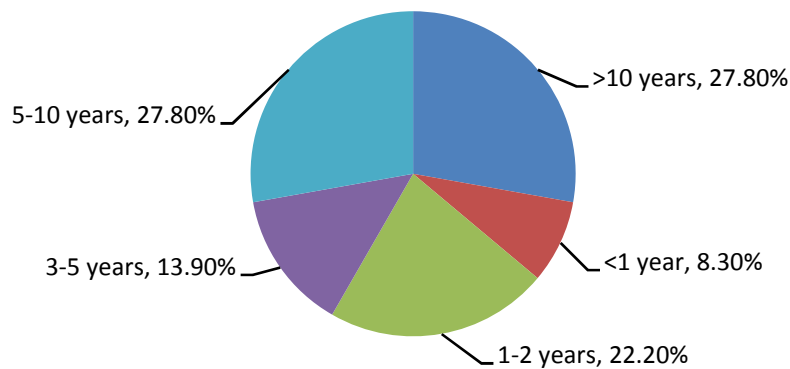


Figure 3: Pie chart showing respondents' experience with SPLE

Figure 4 shows the distribution of responses for the role played in their product-line projects. The majority of respondents claimed having experience as a modeller (~72%), followed by researcher (~67%), and developer (~53%). 39% of the respondents indicated having experience as a team leader. There was only one marketing expert. Open-text responses included architect (~17%), consultant (~8%), administrator (~3%), and project coordinator (~3%).

According to the results from these two questions, the majority of respondents had adequate experience with product lines and had worked as modellers, thus giving us confidence in the credibility of the respondents chosen for our qualitative study.

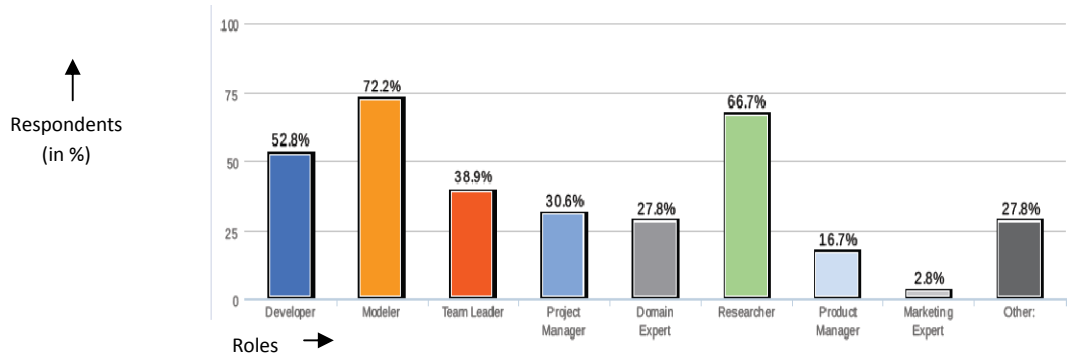


Figure 4: Roles of participants

3.2.2 Contextual Information on Variability Modelling

In order to understand the context of the respondents' experience with variability modelling, we asked about the application domains that the respondents worked on, the types of product-line adoption strategies used in their organizations, and the kinds of artifacts that the respondents had experience with modelling and realizing variability.

3.2.2.1 Application domain

The question on application domain is stated as follows:

"13. What are the application domains of your product lines?"

No multiple-option answers were provided for this question, since there could be many widely varying application domains; instead, a free-text region was provided.

The responses covered a wide variety of application domains. However, respondents sometimes used different wordings for the same domain, or they listed different sub-domains belonging to the same domain. A few of the respondents worked in multiple domains. Therefore, in order to structure the data, we clustered the responses into domains. Table 1 [20] presents a summary of the categorized responses.

The table shows that the top responses are automotive, energy, and enterprise-software domains. The *other* category in the table refers to the set of domains that were reported only once, which includes underwater acoustics systems, geographical information systems, and logistics.

application domain	count
automotive	11
industrial applications and energy	8
enterprise and eCommerce	7
aerospace and defense	5
medical	4
consumer electronics	2
government	2
telecommunication	2
other	10

Table 1: Distribution of participants' domains

3.2.2.2 Product-line adoption strategy

We asked the following question about the types of strategies used by respondents' companies to adopt product-line development. The options provided were based on the three standard adoption methods proposed by Krueger [21], described earlier in Chapter 2 (section 2.2.3):

"12. Which of the following strategies to introduce a product line have you used?"

- a) Product line was developed before any product was derived (pro-active),
- b) A single product was evolved into a product line (re-active),
- c) Multiple existing products were re-engineered into a product line (refactorive),
- d) Any combination of the strategies above

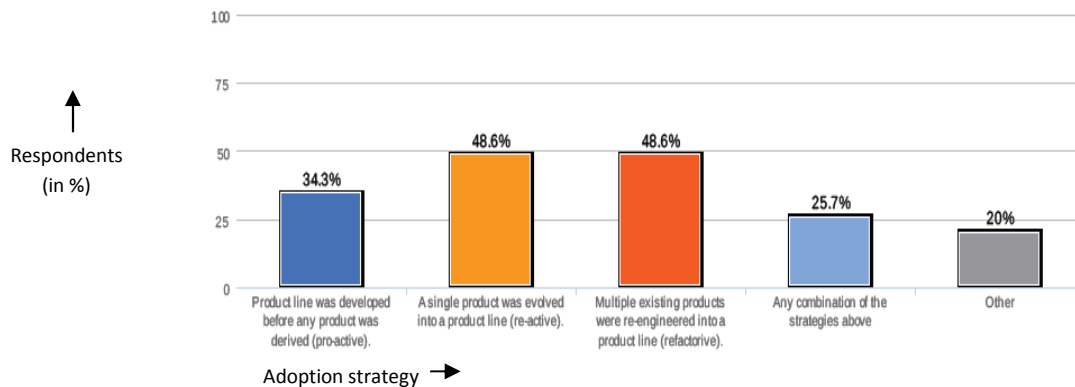


Figure 5: Response distribution on product-line adoption strategies

Figure 5 [20] illustrates the distribution of responses. 48% of the respondents reported that they had applied the refactorive approach, another 48% reported having applied the re-active strategy, and 34% of the respondents reported having applied the pro-active approach. Around a quarter of the participants reported having applied a combination of the three strategies. 20% of the respondents selected the *other* category, which included those who have developed exploratory product lines (e.g. using case studies) or who have not yet completely implemented their product lines. The percentages of responses exceed 100% because some respondents have been involved with multiple product-line projects. The pro-active approach, being the typical product-line adoption approach that involves up-front scoping and systematic platform development methods, is expected to be applied more often [59]. In contrast, our observation confirms that only a small percentage of the industrial product lines are developed using a pro-active approach; one possible reason for most of the companies not pursuing a pro-active approach might be that this approach can impose greater risks, due to a high initial investment, than the reactive or extractive approaches [21], as indicated earlier.

3.2.2.3 Product-line artifacts

We asked about the type of artifacts used by respondents to model variability:

“9. Your models represent the variability contained in which implementation artifacts?”

a) Requirements, b) Architecture/design, c) Platform, d) Components/Modules, e) Libraries, f) Source code (static variability), g) Running product (dynamic variability), h) Test cases, i) Documentation

Figure 6 presents the distribution of responses. The majority of participants represented variability in software components (~73%), and in source-code files (~61%) that model static variations. A good percentage of respondents (~51%) reported modelling variability in artifacts during earlier development stages (requirements and the architecture/design). The *other* category included build files, DSL instances, roadmaps, calibration files, specification models, knowledge representation, and release plans. All these data show that industrial SPL developers find it useful to apply variability modelling both during early stages and advanced stages of product-line development.

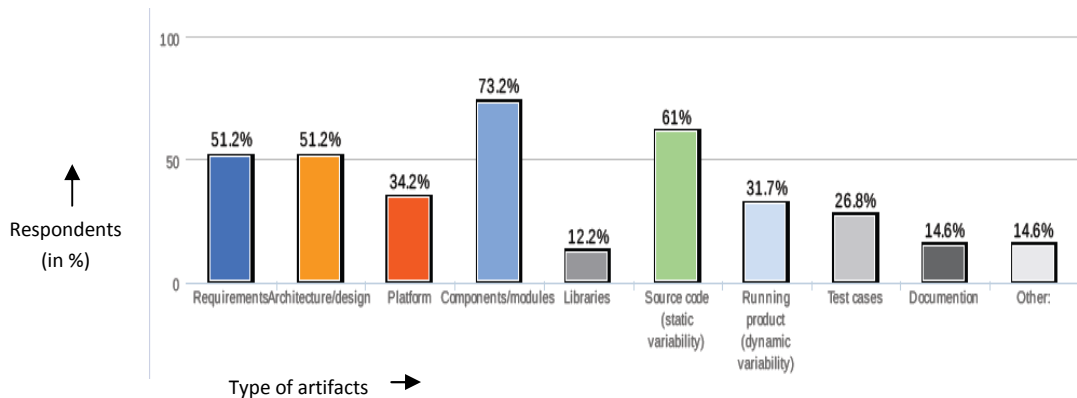


Figure 6: Response distribution on artifacts used to model variabilities

3.2.3 Perceived Value on Variability Modelling

We asked the following question about respondents' opinion on the utility of variability modelling:

"1. Do you consider variability modelling useful?"

a) *Definitely yes*, b) *Yes*, c) *Neutral*, d) *No*, e) *Definitely no*

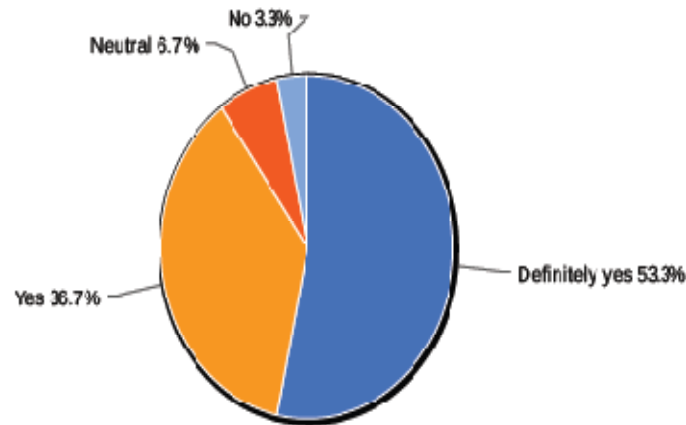


Figure 7: Response distribution on respondents' attitude towards variability modelling

Figure 7 presents the distribution of responses. As shown in the figure, 90% of the respondents responded positively to the usefulness of variability modelling, among which 53% provided a surety by choosing the 'definitely yes' option. Once again, this result shows the credibility of the chosen participants for our variability-modelling study. We did not obtain any open-text responses for this question.

The following question asked about the different uses of variability modelling in the respondents' product-line projects:

"2. Which of the following uses of variability modelling are most valuable in your experience?"

a) *Management of existing variability*, b) *Product configuration*, c) *Requirements specification*, d) *Derivation of products*, e) *Design/Architecture*, f) *Planning of variability*, g) *Domain modelling*, h) *Software deployment*, i) *Documentation*, j) *Quality assurance testing*, k) *Market feature scoping*

Figure 8 shows the distribution of responses. It can be seen from the figure that most of the respondents (~79%) reported the use of variability modelling for managing existing variabilities, followed by a good proportion of the participants expressing value in using variability modelling to support product configuration, requirements specification, and product derivation. Interestingly, a small percentage (19%) of respondents reported the value in using variability modelling to support scoping of features for marketing activities. The open-text responses listed other uses of variability modelling, such as

maintenance, cost estimations of newly added features, and planning of development and evolution. All these responses suggest that there are different uses for variability modelling depending on the product-line organization. This observation contrasts the finding from the earlier work involving a few of our group members on systems domain product lines [86], which established the use of variability modelling to support product-configuration activities.

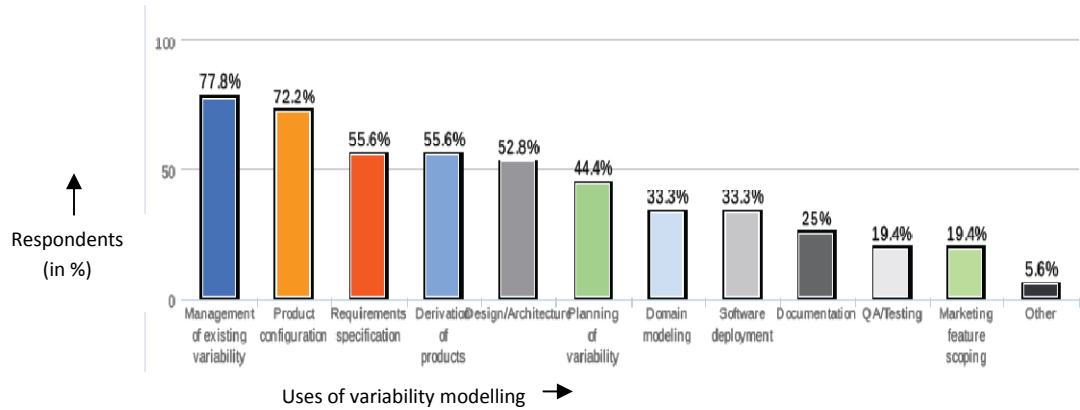


Figure 8: Response distribution on uses of variability modelling

3.2.4 Notations and Tools for Variability Modelling

We asked if the respondents have used separate variability models or annotations to describe variability in their artifacts:

“3. How have you modelled variability?”

- a) *Separate variability models*, b) *Annotation of existing implementation artifacts*

The distribution of responses is presented in Figure 9. The majority of respondents (~77%) have used separate variability models, conveying the widespread use of dedicated variability models among industrial SPL developers. 47% of the respondents have used the annotations. Under the open-text answers, a few respondents indicated using a combination of both representations, and a few of the others listed alternative representations: DSLs, delta modelling, or annotations provided by the Spring component framework [66].

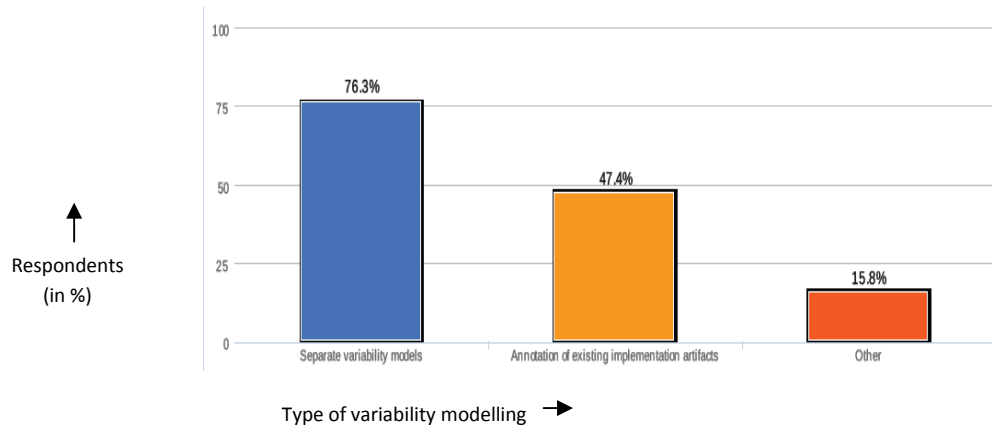


Figure 9: Response distribution for type of variability representation

We asked the following question about the different variability modelling notations used by respondents:

“4. Which notations have you used to model variability?”

a) Feature models, b) Spreadsheet, c) Key/value pairs (e.g. in xml- or text-based configuration or properties files), d) Domain-specific language (DSL), e) UML-based representation, f) Decision model, g) Product matrix

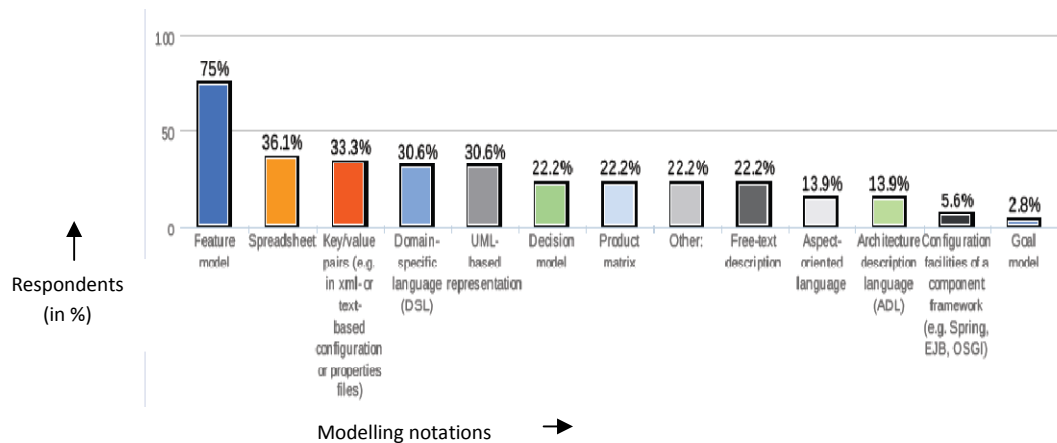


Figure 10: Response distribution on variability-modelling notations

Figure 10 shows the distribution of responses. The vast majority of respondents who reported using separate variability models used feature modelling on their projects. Next to feature modelling, the other commonly used notations are UML-based notations, DSLs, architecture description languages, and a few non-formal representations, including spreadsheets, product matrices, free-text descriptions, and key/value pairs in XML- or text-based property files. The respondents indicated using an average number of three notations for their product lines. Only 8% of the respondents chose one notation.

Interestingly, one respondent mentioned using a total of eight varied notations. The open-text responses included Design Structure Matrices [108] and CVL [48]. All of these different responses suggest that practitioners use a variety of different variability-modelling notations to suit their product-line needs; this observation is also confirmed by one of our groups' earlier publications [86] on the system-software domain. All these responses suggest that respondents used a diverse set of notations to represent variability. There is no solid empirical evidence as of yet to suggest that an ideal notation can satisfy all industrial variability-modelling needs.

We related the responses on variability-modelling notations to the responses of the value-based question on the uses of variability modelling (question 2). The analysis revealed that the respondents who use variability modelling for planning activities apply feature-based notations to represent variability, except in one case. This finding suggests that the coarser and abstract nature of features is well suited for planning activities, in contrast to fine-grained variation points, which are usually used at lower abstraction levels.

We asked about variability-modelling tools that the participants have used on product-line projects:

“5. Which tools have you used to model variability?”

a) Pure::variants from Pure::Systems, b) Home-grown domain-specific tools, c) Other open source tools, d) Other commercial tools, e) GEARS from BigLever Software, f) FeatureIDE from University of Magdeburg, g) DOPLER Tool Suite from University of Linz, h) Product Configurator from Camos, i) XFeature from P&P Software, j) Product Modeler from Configit, k) dslvariantmanagement (open source), l) Oracle configurator/modeler, m) mbeddr.com (open source), n) SAP configurator, o) AHEAD Tool Suite from University of Texas, p) Siebel configurator from Oracle

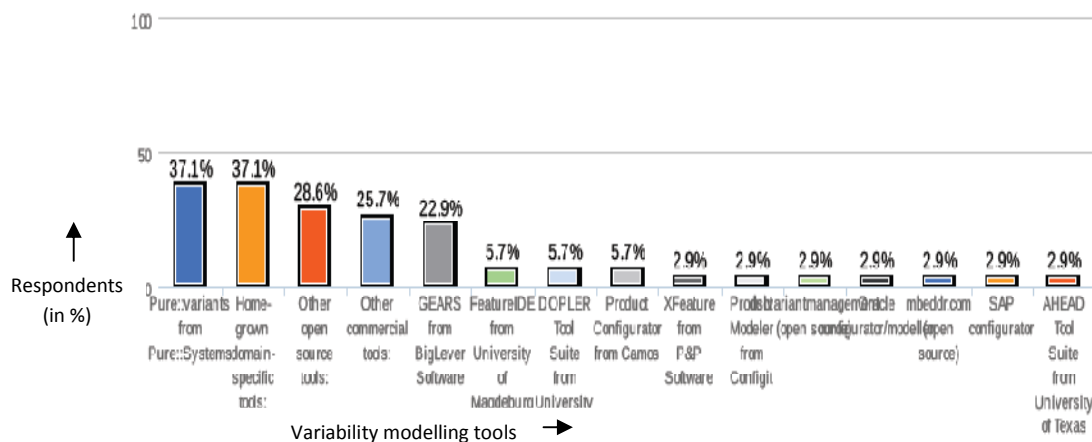


Figure 11: Response distribution on variability-modelling tools

Figure 11 shows the distribution of responses. Note that all of tool options have been used by the respondents in varying proportions, with the exception of Oracle's Siebel configurator, which has not been used by any of the respondents. 37% of the respondents have used pure::variants from pure::systems and 23 % reported using GEARS from BigLever Software. A solid proportion of respondents indicated having used home-grown domain-specific tools (~37%), open-source tools (~29%), and other commercial tools (~26%). Table 2 shows the set of home-grown, commercial, and open-source variability tools used by the respondents; a few of the tools include, Eclipse EMF, IBM Rational Software Architect, Simulink extensions, XML-based representations, decision-model-based Tecnia PLUM [67], v.control [68], SparxSystems Enterprise Architect, and Microsoft Excel. It can be seen from the table that a few open-source variability tools have a research origin and yet are applied to industrial product lines: CVL prototype [48], the CVM framework [69], Hephaestus [70], and SPREBA [71], are also being. These variety of home-grown, commercial, and open-source variability tools shows that industry-specific tools, which are not widely used within the SPL community, are also being applied by practitioners in industrial product lines, despite research-based variability-modelling tools. This observation is also true for our group's earlier study of the systems domain project [86] involving the Linux kernel and the eCos operating systems. It would be worth investigating whether more industrial organizations have developed similar home-grown solutions for satisfying their product-line needs.

All of these different responses on tool usage suggest that there is no single variability-modelling tool or framework that dominates the market. One of the reasons might be because all participants are geographically distributed across the globe (65% Europe-based) and they tend to use or favor tools that are being widely used in their respective region. Another reason could be that the participants in our study are from different industrial domains, and it is possible that different domains may favor different tool choices.

	open-text answer	count
open source	CVL/CVLTool	2
	SPREBA (ADORA)	2
	Eclipse EMF/Ecore	2
	CVM	1
	Hephaestus tool to manage variabilities in SPLs	1
	Spring	1
	Xtext	1
commercial	Enterprise Architect	3
	PLUM	2
	CWAdvisor/ MS Excel	1
	IBM RSA	1
	Microsoft Dynamics AX	1
	systemweaver	1
	v.control	1
home-grown domain-specific	BCS, Dialog	1
	Delta-MontiArc (Architectural Variability), Delta-Simulink (Variability of SL models)	1
	Eclipse-based graphical toolset: FMT	1
	Extension of IBM RSA to model variability in UML	1
	Internally developed tools	1
	MasterCraft	1
	Other internally developed modeling and generative tools	1
	ParameterManager	1
	Various XML, text-tables, source code comments	1
	different tools for different types of variability	1

Table 2: Usage of home-grown, commercial, and open-source variability tools

3.2.5 Scales and Details of Variability Models

We asked the following questions about the units and sizes of the respondents' variability models:

"6. Which of these "units of variability" do your variability models use?"

a) Features, b) Decisions, c) Configuration options, d) Variation points, e) Calibration parameters

"7. How many units of variability (as specified above) do your models have?"

a) 1 model, b) >5 models, c) No model, d) 2-5 models

Figure 12 shows the variability units used by respondents. An average of 2-3 variability units were chosen by each respondent. As shown in the figure, 80% of the respondents reported using features; this is expected, since feature modelling is the most commonly used representation (refer to question 4). Next to features, variation points (~73%) and configuration options (70%) are the most commonly used variability units. 27% of the respondents reported using calibration parameters; interestingly, most of these respondents worked in the automotive domain. One open-text response indicated the use of deltas.

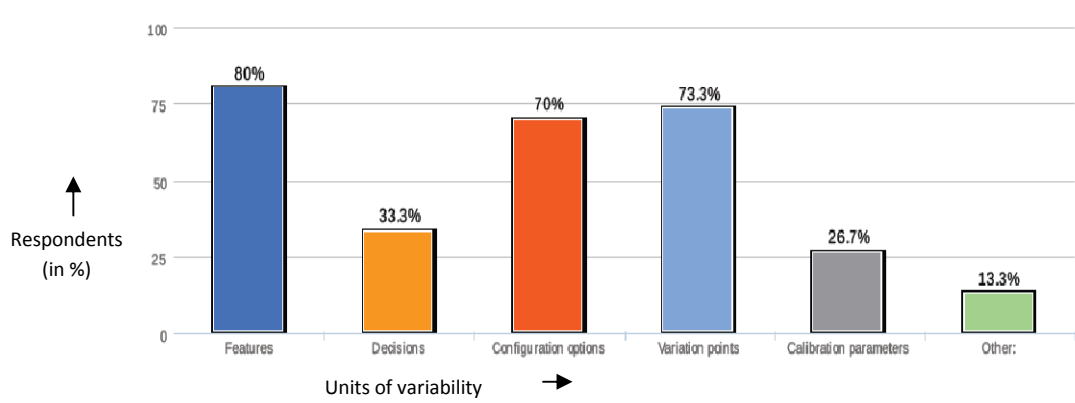


Figure 12: Response distribution on types of variability units

The distribution of responses on the sizes of variability models is presented in Table 3. The columns represent ranges of numbers for expressing variability units (e.g., >10, 000 units, <50 units, and so on), where a variability unit could be a feature, decision, configuration option, calibration parameter, or any other units of variation used by respondents. The rows represent ranges of numbers of models (e.g., 1 model, 2-5 models). Each entry in the table is the percentage of respondents who reported having models with the respective number of variability models in the row and the respective number of variability units in the column. We also asked the respondents to specify their units of variability (e.g., features, variation points). The results in the table indicate that the majority (~69%) responded to using small models, which had fewer than 50 features. Most of the respondents used only a single variability model. A small proportion (~26%) of respondents reported having worked with very large bigger models, having more than 10,000 features; these respondents worked in the automotive, defense, and other industrial domains involving software-intensive systems. We saw that all the respondents who reported having more than

10,000 units, selected either features or decisions as their variability units. Although a relatively small percentage of respondents have used variability models with more than 10,000 units, it confirms that such large variability models exist in industrial product lines; this finding is also established by most of the existing SPL Hall of Fame¹ experience reports.

	<50 ¹	51-100 ¹	101-1000 ¹	1001-10000 ¹	>10000 ¹
0 models	8.6%	22.9%	22.9%	42.9%	40.0%
1 model	40.0%	20.0%	28.6%	14.3%	14.3%
2-5 models	11.4%	14.3%	8.6%	0.0%	5.7%
>5 models	17.1%	8.6%	11.4%	8.6%	5.7%
sum (≥ 1 model)	68.5%	42.9%	38.6%	22.9%	25.7%

¹ units (e.g. features, decisions, variation points)

Table 3: Distribution of responses on sizes of variability models

We asked one question about the usage of cross-tree constraints in variability models. We included a few commonly used relations (such as *requires* and *excludes*) to provide a better understanding of the question among the participants:

“8. Do some of your models have explicitly-modelled feature dependencies (e.g. *requires*, *excludes*)?”

a) 0-25%, b) 26-50%, c) 51-75%, d) 76-100%, e) don't know

Figure 14 presents the distribution of responses. The majority of participants (>75%) indicated that they explicitly modelled dependencies for their variability units. 44% of the respondents indicated having less than 25% of their variability units involved in dependencies, followed by 22% of the respondents reporting on 26-50% dependency involvement, and another 22% on 76-100% dependency involvement. Our findings suggest a less number of cross-tree dependencies being used in industrial variability models; this is in contrast to the high density of dependencies found in our group's study on product lines systems domain [86]. We did not include more specific questions about cross-tree constraints in order to avoid intimidating participants on technical details. However, we feel that an elaborate understanding on the use of cross-tree constraints is essential, since these constraints are typically known to influence the development of reasoning methods or tools [72]. Such an in-depth understanding on the usage of cross-tree constraints is only possible through detailed artifact studies, or through interviews involving industrial participants.

¹<http://splc.net/fame.html>

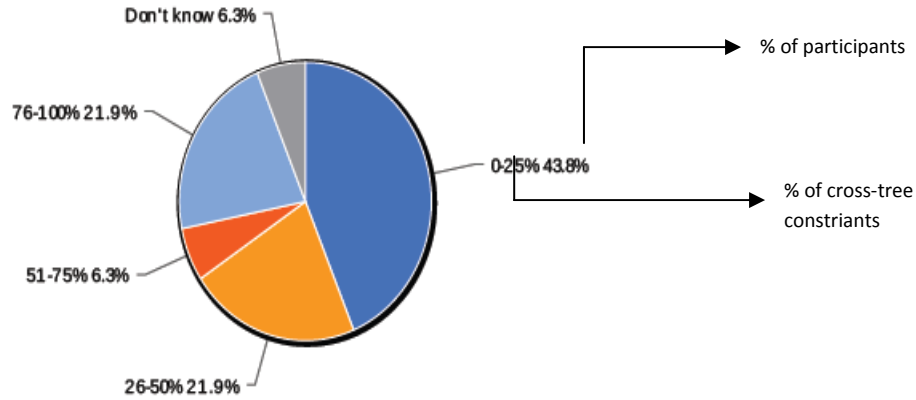


Figure 14: Response distribution on the use of cross-tree constraints

3.2.6 Complexity Challenges and Mitigation Strategies

We asked the respondents about any complexity issues or challenges they encountered with respect to modelling variability, and the respective mitigation strategies used:

“10. Have you experienced complexity problems with variability modelling? If yes, where?”

a) Visualization of models, b) Dependency management (e.g. explosion of dependencies), c) Configuration process (e.g. with conflicts during configuration), d) Model evolution, e) Traceability

“11. What mechanisms have you employed to combat complexity in variability models?”

a) Decomposition into multiple models, b) Hierarchical organization of multiple models, c) Some notion of encapsulation/interfaces between multiple models, d) Abstraction/simplification of variability (hard restrictions on the level of granularity for representing variations), e) Visualization of models, f) View-based editing and visualization, g) Automated reasoning tools (e.g. to check consistency, resolve configuration conflicts or propagate choices)

The distribution of responses on the complexity problems is presented in Figure 15. It can be seen from the figure that the majority (61%) of the respondents reported issues related to visualization of models. Next to visualization problems, an equal proportion (58%) of the respondents reported issues related to dependency management and model evolution. Most of the respondents indicated an average number of 2-3 choices, and a small proportion (~11%) of the respondents chose all the provided options. Among the open-text answers (~15%), the main challenges mentioned were associated with modularization for multiple product lines, tests, and model reduction. One of the respondents presented a challenge that stated: “getting developers to understand why we do this (refers to variability modelling), and the correct patterns to use”.

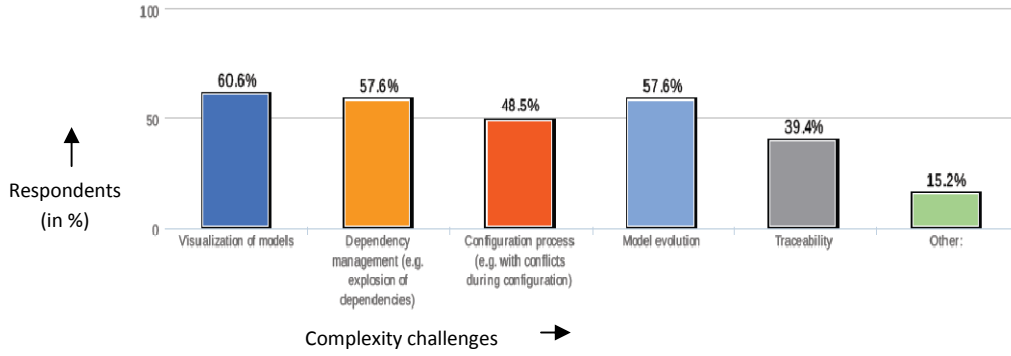


Figure 15: Response distribution on complexity areas in variability modelling

The response distribution on mitigation strategies is shown in Figure 16. A good percentage (~64%) of the respondents reported using hierarchical organization of multiple models for tackling their variability-related issues. Since the majority of respondents used only one variability model (refer to question 7), it is possible that the respondents might be referring to the intra-model hierarchical structure of a single feature model, rather than inter-model hierarchy that involves multiple feature models. 52% of the respondents reported using model decomposition, followed by 46% responding to the use of automated reasoning tools; both of these mitigation strategies are also reported in our earlier work [86] on systems domain. 38% of the respondents indicated using methods for simplifying or abstracting variability, and 33% responded to using visualization techniques to manage their complexity issues. All these different responses suggest that participants use a variety of mitigation methods based on the nature of their complexity problems. Among the open-text answers, one respondent stated the need for variability models to be managed by a small centralized team, since variability models tend “to be very fragile”. As a recommendation, this respondent advises on “*assigning configuration/variability dependent tasks to a small selection of people*”. A similar response is described by another participant as “*application of variability modelling to smaller scopes of development (instead of across entire development effort)*”. Another interesting open-text response suggested that variability modelling demanded “much *manual work*”; he recommends “*rule engines for consistency checking and value propagation (no SAT solvers)*” for reducing the modelling effort.

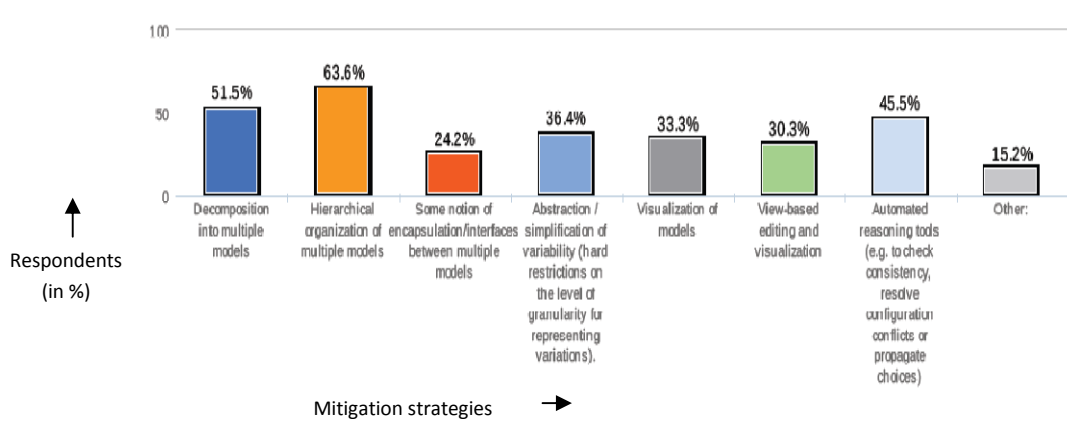


Figure 16: Response distribution on mitigation strategies

3.2.7 Additional Comments by Participants

We added a provision towards the end of the questionnaire for participants to provide their general comments with respect to variability modelling and product lines. A few interesting comments are discussed below.

One participant, with more than 10 years of product-line experience, described the need to explore the capabilities and use of dynamic programming languages to model variability. The participant supported his claim by discussing the variability-modelling techniques in the Java language, as described below.

“Both the field and this study could use a broadening of perspective. My day job is building Java-based server-side software, which tends to be one-of-a-kind, non-product-line type software. Java is a rich language and technologies such as SPRING and MAVEN provide very rich variability tooling. Additionally, using things like PUPPET for deploying into cloud architectures as well as staging and testing infrastructure means that we have a lot of variability. We deploy in different configurations to different data centers, use feature flags as well as AB testing to test new functionality, etc. My feeling is that the research field still assumes a traditional low-tech embedded-software perspective, where the lack of a lot of things need to be compensated for with variability-modelling tooling and cumbersome build systems. So, I don't model variability. Instead I make software that has variation points that are explicitly configurable. The activities of developing and designing when following a continuous deployment model are inseparable.”

There exists a body of SPL research that looks at dynamic product lines and different dynamic language-based techniques, such as those indicated by the participant. However, the modelling of run-time variability still needs to be explored and studied, possibly with adequate empirical evidence. In addition, SPL architectures need to be adapted to support various dynamic variability-modelling techniques, which might be challenging in the case of complex software-intensive systems.

One of the participants commented on the usefulness of qualitative studies, such as ours, about commercial variability modelling. However, another participant expressed doubts about the integrity of the provided multiple-option choices and our chances of being able to draw any solid conclusions due to the differences in interpretations that might arise among the participants. Along similar lines, a few respondents expressed difficulty in interpreting the questions about the sizes and structures of variability models. A few example comments are reported below.

Example comment 1:

“Difficult question about the size of models, as it is unclear how to treat hierarchical composition of models as one or multiple models. In general: Decision Models and Feature models in early phases (requirements, tendering, product planning): up to 1000 elements, for configuration “models” (static or dynamic parameterization) usually >10000”

Example comment 2:

“I hope I got the question “How many units of variability do your models have” right. I didn't really understand the reason for the multiple columns. Right now we have 1 model with <50 variations, but we are very early in our implementation of PLE.”

3.3 Threats to Validity

Our findings are solely dependent on the declared information provided by the respondents through the online questionnaire and therefore are subject to the usual validity threats in a questionnaire study. All responses depend on the perspectives and beliefs of the survey participants and their interpretations of the provided questions and answer choices. Hence, it is possible that the responses may not reflect actual organizational practices. We have compensated for this threat to some extent by requiring the participants to disclose their name and organization and by removing the responses from pure researchers. We were able to confirm the validity of the respondents' affiliations. It is also possible that the participants might have misunderstood parts of the questionnaire, given the inherent inconsistencies in terminology that exist in SPLE literature. In order to evaluate the responses on the value of variability modelling, a Likert scale was used. The Likert scale indicated that the majority of respondents acknowledged the value of variability modelling (55%- definitely useful, 35%-useful, 2 neutral opinions, and 1 negative response). We could not draw any solid conclusions or correlations because this study was limited to a small dataset. However, this questionnaire study helped us to select participants for the second part of our qualitative study: semi-structured interviews. The interview results are discussed in Chapter 4.

Chapter 4

Qualitative Study: Part 2 - Interviews and Results

This chapter presents the second part of our qualitative study, which involves a set of interviews on variability-modelling practices and product-line adoption in industry. We first describe the methodology and the sources used for selecting the interview participants. Then, we present a detailed description of the interview results. We conclude with a comparison and discussion of the variability-modelling and product-line adoption practices, based on the interview data.

4.1 Interview Design and Participant Sources

We conducted seven semi-structured interviews, among which two interviewees are employees from the same company; each of the remaining subjects is an employee involved in the product-line efforts of a single company. Each of the interviews lasted between 1-2 hours. Our subjects include two industrial companies who apply SPLE and variability modelling, two SPLE consultant companies, and two tool builders who develop tool frameworks for the design and development of product lines. Both of the tool-builder companies were chosen based on the widespread commercial use of their tool frameworks. One of the industrial companies was selected because of their highly-referenced experience reports that disclosed their product-line efforts. The other industrial company and the two consultant companies were chosen based on their employees' responses to our survey on variability-modelling practices (refer to Chapter 3); we emailed interview-invitation letters to the participants from these three companies.

The two consultant companies are medium-scale (50 employees) companies that offer consultancy services and develop custom software solutions for their clients. The two industrial companies are large-scale companies (>23,000 employees) that develop software product lines for their main products. Each of the tool builders owns a tool chain, along with a set of extensions, for developing product-line management technologies, especially for modelling and configuring variability. Both tool chains are being widely applied in industrial product lines. The tool builders are currently focused on inventing solutions to integrate their tool frameworks with existing development processes and tools used by industrial organizations. Table 4 shows the summary of interviewees involved in our case studies.

We prepared an interview guide that covered two main topics:

- ***Variability modelling and realization:***

- Practices regarding product-line adoption strategies and information on types of product lines,
- Information on the processes used for building variability models,
- Interviewee's roles and experience in variability modelling and product-line development,
- Information on variability units, and kinds of variabilities in variability models,
- Characteristics of variability models (e.g., sizes, dependency relations, constraints, and so on),
- Processes for modularizing variability models,
- Information on the types of evolution in variability models, and the strategies used to manage evolution in variability models and artifacts,
- Information on the types of product-line artifacts, and their mappings to variability models.

- ***Benefits, challenges, and recommendations:***

- Perceived advantages concerning variability modelling and product-line adoption,
- Challenges encountered during variability modelling and product-line adoption, and
- Recommendations to improve practices regarding variability modelling

The interviews were recorded, transcribed, translated (in two cases, see Table 4, credit to *Thorsten Berger* of the *IT University of Copenhagen*) and analyzed with respect to the two above-mentioned topics and their sub-topics. We present the practices and experiences gathered from each organization's interviewee(s) in the form of a case study. Although we describe common practices and trends across the organizations based on the interview data, we do not attempt to reach any statistical deductions or generalizations because of the small number of subjects. The practices, benefits, challenges, and recommendations about variability modelling and product-line development are based solely on the interviewees' perceptions. Please note that we sometimes annotate the interviewees' quotations with bracketed interpretations of sentences, phrases, or terms used, to provide a better understanding of what the interviewee means or is referring to.

<i>Case Study No:</i>	<i>Company's Name (used in case study)</i>	<i>Type of Company</i>	<i>No: of Interviewees (per case study)</i>		<i>Language used by Interviewees</i>	<i>Transcript on/ Translation</i>
1	global producer of electronic and mechanical components	large-scale industrial company	1		English	transcribed in English
2	consulting company for web-based applications	medium-scale consultant company	1		German	translated to English
3	IT consulting company for enterprise, embedded and mobile applications	medium-scale consultant company	1		English	transcribed in English
4	automotive company	large-scale industrial company	1		English	transcribed in English
5	BigLever GEARS	tool builder	1		English	transcribed in English
6	pure::variants from pure systems	tool builder	2	interviewee 1	German	translated to English
				interviewee 2	English	transcribed in English

Table 4: Summary of interviewees in case studies

4.2 Interview Results

4.2.1 Case Study 1: Global Producer of Electronic and Mechanical Components

Our first case study focuses on a large-scale company that develops electronic and mechanical components for industrial applications, including refrigeration, pressure pumps, and power electronics. This company owns several software product lines [30]. For this case study, we focus on the company's division that owns a product line of software controllers for power electronics systems. This software-controller product line was introduced in 2009 and was successively extended to its current size of 1.5M lines of code. The product line comprises twelve main products, and over 30 optional add-ons for specific sub-products. The company uses one feature model to describe the commonalities and variabilities of the all of the twelve products and their sub-products in the software-controller product line. All the product-line assets are fully integrated in one platform. The implementation languages used are C++ (98% of the code) and C. The practices and experiences described below are from the

perspective of a software architect who performs variability management for the software-controller product line.

A. Variability Modelling and Realization

Product-line adoption: - The product line's adoption was refactorive. Originally, one product existed that was cloned into a second product, which then adapted and evolved on its own. This cloning continued for several other products, simply by branching with the version control system. After three years, an effort began to re-engineer all products into one platform. The developers used a textual diff to compare the first two products, merged their commonalities to form a unified code base, and realized their variabilities by introducing variation points using the C/C++ preprocessor (#IF and #IFDEF directives). This process continued for the other products; however it was challenging to reach a full integration for the reasons described by the interviewee:

“The first 50-60% were easy with less effort, but the last 20% was more challenging because it could not be achieved by using simple diff because of some kind of logic difference, difference in the semantics, the algorithm changes. Or there was some extra functionality added, so, it was somehow not easy; it was a manual process, and took a lot of time.”

Creation of variability model: - A dedicated team is responsible for variability modelling and management. This team works with different development teams to build the integrated platform that supports all products. A domain expert builds the feature model and ensures its integrity and correctness. The interviewee indicates that they prefer to assign only one person (usually, the domain expert) to be responsible for the construction and maintenance of the feature model. However, when there are modelling issues, or if the domain expert needs clarification regarding some modelling aspects, there are meetings or workshops with development teams to discuss them.

“We have an expert who has great domain knowledge because he took care of the development and all this, and then he does consult model development teams. So, we try to have one person who is responsible, but he does not alone decide all the things; whenever we have issues, we have meetings to discuss it and he (domain expert) is responsible to make models correct. So, he consults with other teams, for e.g., we assumed that we model main products pretty well, and now we have options to model and then he (domain expert) was telling me that we have more than 30 options or 30 sub-products and not all are options, and then he consult individuals responsible (for validating the options)...”

Modularization and structuring of feature model: - There is a single feature model that is shared among various development teams across the organization. There was an interest in modularizing feature models, such that different modules represent different kinds of products in the product line. However,

the modularization attempt was disbanded because they could not find a good modularized structure that would accommodate all of the different feature combinations associated with product types.

“... because we have kind of different products in the product line, and it is possible to split it. We actually find it difficult to find a good structure because we have features that are common and those that are specific to certain product kinds, types, but also commonalities between two groups; and then if we kind of follow all these combinations, we are not kind of not going to make it simpler, but we are trying to create some kind of logic bundling of feature, but. . .”

The feature model contains mostly functional features. Interestingly, bugs and bug fixes are occasionally represented as features. Some bugs became natural behaviours to some customers, and not all customers want to have them fixed.

“...we have different defect-fixes, and people would not enable them, so a bad defect or fix-a-defect is not integrated to the source code, we somehow put it as a new feature. Sometimes, our customers also have these features; so, even though the behaviour may be not the intended one, they are used to this. So, we have to be careful, more like if you find a fix in one product, may not somehow be in another...so, you know, we have to be really careful with it...”

The interviewee mentioned that the company uses one feature model to represent the variabilities and commonalities in all of their main products and sub-products. Their feature model consists of 1100 features, among which 800 are leaf nodes and the rest are high-level features used to categorize functionality. A feature may have anywhere between 1-20 sub-features. The interviewee uses different strategies to organize the feature model, for example combining product-specific features in one tree branch and common features in another branch. The height of the feature model is usually restricted to four levels, as a result of the company's decision to maintain simple feature models. However, keeping a feature model simple is not always easy:

“We don't like this great depth; we try to keep it simple, but sometimes, it becomes too hard somehow, and keeping the number of features becomes difficult.”

Feature dependencies, data types, and cross-tree relations: - Most of the dependencies in the feature model are traditional hierarchical dependencies, especially *mandatory* and *optional* relations, and feature groups, especially *OR* and *XOR*. 95% of the features are of Boolean type and they are used to switch on and off functionalities. In some cases, integer features (16 and 32 Bit integers), string features (e.g., version numbers or strings to realize menu differences in controllers), or more sophisticated types (such as data structures, which are cast to strings for compilation) are used to express variant options. Explicit default values are specified for some features, and there exist default configurations for the main products. The main cross-tree constraints used are binary relations, especially *requires* and *excludes*. The

cross-tree relationship *recommended* is used to convey the preference of choosing one feature over another.

Cross-tree constraints are intentionally used less frequently because of the interest in maintaining simple feature models that are easier to view and reason.

Mapping to artifacts: - Presently, the company maps only code-based artifacts to feature models. The interviewee indicated that they are interested in mapping requirements to features in the future. The feature models are constructed using pure::variants, and the mapping between features and code artifacts is realized with pure::variant's concept of a family model. The family model reflects mainly variable code-based artifacts (especially a file system, which contains more than 10,000 source files). Files contain up to 70 features. The interviewee said that his company used #IFDEFs extensively, probably around 14,000 #IFDEFs among 10,000 files. Interestingly, the interviewee reported that the use of #IFDEFs affects code readability when the number of features is large:

"We started looking how to include situations where we have files that have 70 features or #IFDEFs. I mean, among 10,000 files, we have 14,000 #IFDEFs that are used in 14,000 cases, and that create a challenge for developers to read the code."

Furthermore, dependencies between the 10,000 files are not modelled in the family model, but instead are reflected as dependencies between features and their mapping to files. The interviewee reported that that some features map up to several hundred files. This observation is interesting, since earlier it was mentioned that there are few cross-tree constraints among features in addition to hierarchical dependencies.

"We try to keep it simple, but in the family model, with the file system, there will be restrictions more complex, especially like the linking of files to features, like a file could be linked to many features, like include those files and maybe that extra file....so, it's complicated, the conditions..."

The concept of configuration space in pure::variants is used to configure products in terms of feature selection. The mappings between features and artifacts in the family model are realized by feature makefiles that describe what artifacts are selected when a feature is enabled or disabled. In addition, each product in the product line is associated with a configuration file that contains parameters, which are set using #DEFINE preprocessor directives to either enable or disable features. All configuration files are maintained in a central database. The binding-time for artifacts is compile-time; run-time variability is not supported.

Evolution of feature model: - New features are added frequently, and a few features are removed from time to time. Changes to existing features are difficult, and are not encouraged. The growth-rate of features is not linear: the number of features increases between 5–10% per year, and 3–4 stable product

releases are launched every year. In the interviewee's opinion, although there are clean-up tasks performed periodically to remove unused features, not effort is expended on removing unused features. The interviewee suggested that it would be better to have a standard procedure for removing enough unused features, since the size of feature model is growing considerably. The feature additions and removals have not affected the existing hierarchy of their feature model.

During the periodic clean-up tasks (performed for removing unused features in the feature model), the consistency between the evolving feature model and the code artifacts is checked using code inspections and reviews; this helps the developers to improve their quality of code artifacts. Around 98–99% of the changes to the software are reviewed, and those features which are not enabled in products are removed.

The interviewee points out that colleagues have started investigating the use of DSLs to configure sub-components of the product line. Currently, there exists one DSL for configuring a specific sub-component; this DSL is partly configured using features from the feature model.

B. Benefits and Challenges

Management is satisfied with the adoption of product lining and the application of feature modelling, and it acknowledges the value of feature models. In contrast, the developers find it difficult to cope up with the adoption of product lining and the transition to feature models, because they are used to working at the code level and they are now forced to think at the modelling level and in terms of reusable artifacts.

“It's that developers are used to work for a long time on the same abstraction level, basically text. But now somehow, we introduce a new way of working because we cannot merge everything at the source code level, but also need to think at the model level, for e.g., merging artifacts: it is not enough to merge the source code, but also have to consider merging the models they developed. So, whenever they add a feature, they add the feature to the model, so, later whenever they do merge back the integration branches, they have to merge all the artifacts.”

The developers do regard the use of pure::variants as an advantage, because dependencies between features and artifacts can be better managed. Feature modelling also helps the developers to visualize variations at the level of features. For instance, the developers were able to identify duplicate features, which were not detected previously. In addition, feature models help the developers to better understand the code in terms of features.

“Now, we can see common features that are shared, and also explore the relationships, especially because we can see what configurations are allowed and not allowed, and these relations were not easy to explore at first; it is also important because before, we noticed that the sent functionality was

implemented twice within the same project basically, which was not visible at first... Now you can understand the code easily, you can see the difference in features between the products.”

Other product-line contributions reported include a) improved quality of software, for example by reducing the number of critical software bugs, and b) a significant reduction in the time-to-market for products. Currently, there is tremendous interest in further shortening the time gap by providing support for modelling product-line and portfolio requirements, as well as their dependencies.

Despite these advantages, the adoption of product-lining introduced a few challenges for the organization. Product lining forced the developers to test components individually using unit testing, rather than depending on their earlier practice of testing the products as a whole. This posed a difficulty in some cases to encapsulate components and test them in isolation for run-time environments.

According to the interviewee, there is a struggle between developers and management, and it has become difficult to enforce practices. For instance, product-line development calls for a new organizational structure, and eventually requires changes in the thought processes and working patterns of developers. The developers are forced to think in terms of creating reusable artifacts. Our interviewee considers it effective to dedicate a development team to domain-engineering activities. However, this organization can give rise to strained relations in the company when the three main levels of authority: the product management responsible for the business and economic objectives; the technical people responsible for the product development; and the middle management, have difficulty in reaching consensus:

“...introduction of a product line requires some kind of organizational structure, and this introduces change, people will have to start thinking in terms of developing assets that can be reused, and this is only possible if there is a group that take care of domain engineering....., this is a strained situation, when we have product development that really looks at the business and the economy point of view, and then you have the tech people that pushes for doing the things the right way and then the management in between. We have to earn money, it is nice to have it but does not apply in all cases, for example you have a product, develop it, and then you want to forget about it, it’s easier to do a clone-and-own, but in our case we have a product and we have to support it for more than 10 yrs and we cannot afford to do that.”

4.2.2 Case Study 2: Consulting Company for Web-Based Applications

The subject of this case study is a consulting company that develops highly customized web-based e-commerce and enterprise applications. This company uses PHP at the back-end and HTML/CSS at the front-end to develop most of their applications. Their software-development process uses generative programming, component-based, and product-line engineering methods to automatically generate reusable artifacts. This company has developed one single product line for web-based applications, and

they use different variants of this product line to develop applications for their web-based clients. The product line has been in production for more than 2-3 years. The interviewee is a developer who works on the software generator, the target code, and the feature model for the product line.

A. Variability Modelling and Realization

Product-line adoption: - This organization followed a re-active approach to build their product line. Initially, they researched and gained experience with product-line processes, using a minimal version of a sample webshop domain that consisted of a small number of features. A generator was first developed for this webshop domain, and it was iteratively extended until it accommodated all e-commerce applications for all of their customers.

“We have one generator for one product line. It was the start of (refers to the company name) in 2003, when we founded the company. Web shop was the sample domain that we chose for also doing some research with product lines and getting experience. We built a second generator, but I wouldn't call that a product line. It was ring manufacturer who wanted to have a ring configurator. We built a generator to generate the 3D models of these rings... The first experiment was driven by practice. We had this web shop in a minimal version and converted that into the generator-based approach. We iteratively extended it.”

The developers attempted to create the target code in parallel with the generators, but deciding how to realize variability in the source-code artifacts slowed down the process. As a result, they used a systematic and interactive approach whereby a generator for configurable software components was built first, followed by the development of the reusable source-code components. In the interviewee's opinion, a parallel approach would be beneficial for a larger development team, since it separates the development of the generator from the development of the target code; whereas theirs was a small team comprising only two people.

“We tried to develop the generator and target code in parallel (based on theory from literature). However, we noticed that the second case (means the parallel approach) slows us down, because when you work in both worlds and you come to a spot in the target code where variability is addressed, you automatically always ask yourself whether that's something that you resolve in the target code or does it have to be in the generator. Then you start to ponder what makes the most sense and you lose time... We were a small team: two people, there was no question of further organizing the teams into platform and product developers. However, in general, we think that there should be target-code developers that only develop and maintain target code; and generator guys, who transform all the stuff and know how it looks like in the generator. . .”

Creation of the variability model: - The features are identified after obtaining a thorough understanding of the domain and the requirements from customers, through meetings or discussions. A feature model was built for the identified functional (or customer-visible) features, in such a way that the features are easily identifiable and selectable by the customer.

“We internally sat together, since we knew some requirements of the customers. We knew what the shop should be able to do. Based on that, we structured features where we thought they make sense; features that customers should be able to select... based on the customer's perspective. We thought about what we can sell, what a customer would like to have. Some commercial thinking, not like (refers to interviewee in case study 5) did it (asking why, why, why), since we internally created the feature model.”

The scoping of features and the construction of the feature model are heavily influenced by the product-line methods proposed by Czarnecki and Eisenecker [24]. The developers initially structured the features hierarchically, starting with the most abstract features. The hierarchy was then modified by extending or refining the abstract features. The interviewee described this process for creating the variability model using an e-shop example:

“For example, imagine a shop system. We had a feature called Catalog System, which was the basis, since a shop always has a catalog. Under this feature, we put features like Shopping Cart. Only when you have a catalog, it makes sense to have a shopping cart. Then you continue with stepwise refinement, extending your features.”

The interviewee indicated that the feature model is used only internally within the company to help their clients build their product-line applications, and is not directly accessible to the clients. The feature model mainly helps the developers to model and configure variability and to derive products. Except for the root feature, commonality is not usually modelled in the feature model. The variability-modelling tool used is CaptainFeature [79]. In the interviewee's opinion, this tool was the best available option at the time, and the tool had a relatively better user interface compared to the existing variability-modelling tools for feature-model construction. In addition to CaptainFeature, the developers use a secondary variability- management tool proposed by Patrick Otto, which is an Eclipse plug-in. Although this tool offers better zooming capabilities compared to other existing tools at the time, there are some visualization problems as the number of branches in the feature model increases. From the interviewee's perspective, the variability-modelling tools currently being used by the company are not optimal, mainly due to the limitations in expressing cross-cut dependencies between features and in detecting or resolving contradictions among features.

“We didn't really have support from the tool where we could see that feature X collides with feature Y. There's nothing like an intelligent inference engine. Sometimes, we reached a point where we didn't know what the feature does or what happens how, or when the nesting was too deep. . . ”

The interviewee mentioned using DSLs to model and configure variability in some domains. For instance, the interviewee talked about a ring manufacturer who used a ring-description language to model the domain entities and their variabilities, and a DSL-based ring configurator to generate 3D models of rings.

“We built a generator to generate the 3D models of these rings. It can be the case that we were at more than 100 features, because the rings were quite fine-grained to configure. But, it was less the domain-specific aspects, it was a ring description language, geared towards the end system, not abstract, more concrete; DSL with domain-specific keywords. Had more than 100 of the main entities (similar to features). We won't talk about this product line in the remainder.”

Modularization and structuring of feature models: - There is a single feature model that consists of approximately 40 features and accommodates all of their customers' features. The height of the feature model is between 5-6 levels and the number of children for each feature varies from 1-6. The interviewee indicated that the feature model has been manageable until now, despite the addition of new customers. We do not have information the company's interest in modularizing their feature model into sub-models.

Feature dependencies, data types, and cross-tree relations: - The main data type used for features is Boolean. The interviewee is uncertain as to whether integer or string features being used. Default values are not supported for features. The main feature dependency relations used are *mandatory*, and feature groups, especially *OR* and *XOR* groups, where the *OR*-relation is used most frequently. Cross-tree relations between features are not explicitly represented in the feature model; however, the interviewee said that the developers are responsible for being aware of these constraints during product derivation.

Mapping to artifacts: - Variable artifacts include HTML pages and CSS, database schemas, PHP scripts, and Java code. Variations in the artifacts are realized by frame technology, based on their own preprocessor version of XFraser [80]. The mappings between the feature model and the artifacts are captured using Java-based imperative scripts that reside within the product-line generator. Artifacts are bound at compile-time; run-time variability is not supported.

Currently, there is a strong interest in establishing traceability between variable artifacts and features; the interviewee said that this would help to assist the developers in understanding the implications of feature selections on product-line artifacts during product derivation. As a first step towards establishing traceability, there is an ongoing attempt to associate more details to features in the feature model, especially for linking test cases to features.

The developers made multiple concerted efforts to use EMF to develop the product-line generator, but they eventually gave up because of certain shortcomings with the framework, including poor readability of the generated classes and difficulty in supporting feature variations.

“We experimented with EMF, we thought it might be an ideal base for our generator framework (since they were already using event system of EMF and Eclipse); but the overhead of EMF was too high. Adaptations are too difficult. We really tried for quite some time. A while ago, we again tried to use EMF for our generator framework, but the generated classes contain too much hard-to-understand stuff.”

There was early interest in developing a product-configuration tool that can be configured by customers according to their interests; but this idea was dismissed because customers often did not understand the functionalities of features and needed guidance in selecting features.

“When we began in 2003, we thought about building a tool for end users, where the user can freely configure a product. We didn’t realize, it’s still quite difficult for a customer, to understand what the features do. When I select feature X, what does it do at all? We noticed that, even when the domain was with very domain-specific terms; in the end you need a consultant to tell him what he needs at all.”

Evolution of the feature model: - Feature addition is the only type of feature-model evolution, which occurs when a customer requests a new functionality. Features are added at the rate of 2-3 features per customer. Additions do not typically affect the hierarchy of the feature model, and the feature tree is re-structured occasionally, especially when the number of features grows significantly. The interviewee indicated that evolution management is one of their current challenges, which has not yet solved (see the next section B for a description on this challenge).

B. Benefits and Challenges

According to the interviewee, the adoption of feature-oriented product-lining has helped the developers to scope their clients’ features better. In addition, feature modelling has helped the developers to visualize features and their relations, and to obtain a better understanding of the different variable options.

“Having a tool and management support to see what capabilities/possibilities my product line has and which customers have which features and their relationships. And to visualize all that information, the variations . . .”

The interviewee expressed mixed opinions regarding how much product-line adoption had reduced the time-to-market of their products.

“Of course, it shortens the time-to-market for individual products, in particular when just making small changes for a customer. On the other hand, I cannot make it in an unplanned, or not well-elaborated

way, otherwise I break something in my product family, which would not be planned. Of course the time-to-market converges to 0 if the customer wants something that we have already realized. . . “

As mentioned earlier, the biggest challenge that the company currently faces is managing the evolution of the feature model and the product-line artifacts. As the feature model and the artifacts evolve, there are no methods currently available to ensure the consistency and integrity of the models and the artifacts and the mappings between them.

“I think the biggest problem we faced at that time and also today (and which is not solved yet) is the evolution. That is, to know the mapping of features to code. To not break anything, in particular when working with or evolving the generator. Further traceability, and making features and their dependencies visible to the developer. Yes, to synchronize code and model and to check that it still fits together is a bigger issue. . . ”

4.2.3 Case Study 3: IT Consulting Company for Enterprise, Embedded and Mobile Applications

The third case study involves an independent IT consulting company that assists clients with software development for enterprise, embedded, and mobile applications. The company offers expertise in product-line research, and sells methodologies and technologies for building product lines. The company is currently involved in two projects, which include a business product line for insurance applications, and a product line for a refrigeration system. This company mainly uses DSLs for expressing variability in most of their product-line projects. Feature modelling is used in only two projects: the insurance project and a Siemens project (AMPLE) for building home-automation systems [81]. The Siemens project was an in-house research project only. The practices described below are mainly based on the product-line experiences of one of the company’s consultants who worked on the insurance and the refrigeration projects.

A. Variability Modelling and Realization

Creation of variability model: - The insurance project uses feature modelling to specify and configure variability at the design (component) level; domain-level variations are not modelled. For this project, there were two or three architects who were responsible for constructing, maintaining, and configuring the feature models.

The insurance project initially used textual DSLs to represent and configure variability, and the project developers faced major challenges in expressing variability. For instance, they had difficulty expressing different types of variations, such as variations in component interfaces, variations at the architectural level, and so on. In addition, the relations between the variations were not easily understood by the developers. In this context, the interviewee introduced feature models to assist with the modelling and

realization of variability in software artifacts. Although, the interviewee did not participate in the feature-identification process, he assisted the clients with their feature-model construction, tooling, and product-configuration activities.

“They knew that they had variability issue, and so what I taught them specifically was the integration between the textual DSL they used for describing their components, interfaces (parts), all this architecture stuff, and the connection of that to the feature model. So I mean, using feature models in a say, naive or simple way, is not very complicated. So, they basically figured out how to describe the variability (using the feature model), especially since the variability was about: do you use this or that component. So, it was not very domain-oriented so to say, they did that, and I helped them with the tooling and the integration of the two worlds.”

Variations in the insurance product line were handled mainly in rules that captured the business logic. These rules were encapsulated within configurable software components. With the help of the interviewee, the developers described the variabilities in their components and captured them using a feature model. Unfortunately, we do not have information on the process used by the developers for building the feature model. The features in the feature model mainly represented functionalities of components and their variabilities, rather than domain-level variations. Hence, the features were not customer-visible features, but were rather technical or component-level features. The tool used for feature-model representation and configuration is pure::variants.

“Customer (refers to the client’s name) is a company dealing with insurance...they used it (feature model) to describe variability of insurance applications, but not on the level of business logic, like this contract has this rules. But, rather they encapsulated some of the logic within, what you called components and then they express variability over which components are used in which product in the feature model. This was something what technical people did, not the domain experts, there was no domain modelling....

So the features would be mostly functional things mapped right away to sort of components, it was really like using this component or that component, so it was functional stuff. Component variability is the main, if the insurance system had some, you know, support for a bunch of different or additional calculation components they know how to deal with. . . ”

The refrigeration project did not use feature models because the currently-used DSL was adequate enough to express all their variabilities.

“Right now in this (refrigerators) project, we don’t use feature models since we do everything with DSLs at least at this point. Currently, we have decided not to introduce other tools just for expressing, at least in this case, for a relatively low numbers of this kind of switch-oriented configurations.”

Modularization and structuring of feature models: - The developers in the insurance project used one feature model, which is developed, maintained, and configured by two or three architects, who work as a team. We do not have information about the strategies used by the company to organize and manage the structure of their feature model. We do not know as to whether the company is interested in decomposing their feature model into sub-models.

Feature dependencies, data types, and cross-tree relations: - The insurance project's feature model has fewer than 100 features and a height of 4 levels. Most of the features are of Boolean type and they are used to represent the presence or absence of components. The parent-child feature dependencies used are *mandatory*, *optional*, and feature groups, especially *XOR* and *OR* groups. A limited number of cross-tree constraints are specified between features, which mainly reflect the *requires* and *excludes* relations.

Mapping to artifacts: - For the insurance project, the main product-line artifacts are software components and their implementations. Family models are not used to specify and configure the artifacts. The interviewee said that family models were not required, since the developers associated the feature model to an architecture DSL. Not much information is available regarding how the feature model is linked to the architecture DSL. The interviewee indicated that there is no explicit mapping specified between the artifacts and the feature model.

"We didn't have a family model actually the family model is interesting because it basically represents the implementation structure . . . (they did not use any model to represent the implementation structure), and in case you do it with DSLs, architecture DSL, and by connecting this (DSL) to the feature model, we don't need the family models."

We do not have information on the variability-realization mechanisms, binding time, or the evolution of feature model used in the insurance project. We did not discuss these aspects for other projects, since they were DSL-based.

B. Benefits and Challenges

According to the interviewee, the developers and the architects in the insurance project appreciate the visual and the configuration capabilities of feature models.

"Well, feature models basically are a nice way of organizing Boolean configuration switches and we used it for that. So, it's easier to get overview over the very things you can turn off or select or unselect or exchange, as opposed to take a bunch of DSL files and scanning through them and see basically the #IFDEFs, if you will; it is just a summary notation of the conceptual variability you have. And obviously, potentially if you have many places that depend on the same switch figure, then obviously you have your traceability by connecting all this different implementation things to the same feature."

We asked why feature or variability models were not used in other product-line projects. His first reason is that some variabilities are better expressed using a DSL, for example certain rules or algorithmic variants. Secondly, medium- or small-scale companies cannot easily afford variability-modelling tools such as pure::variants. Finally, the existing development teams are not always willing to adopt feature modelling because it calls for drastic changes at technical and organizational levels.

“Three main reasons why people don’t use feature models, first is, certain variability can’t be reasonably expressed, let’s say define custom business processes or cooling algorithms for refrigerators, you can’t describe behaviours and expressions with feature models, so we use DSLs. The other one is pure::variants costs money and that is, if you specially live in an eclipse world where everything is open or free, then having this one product which cost money is a very tough sell. And the third thing is that, feature modelling really is specially used for, if it applies on very different levels like requirements, design, etc., and nobody was willing to do this crosscutting change of organization work.”

On a related remark, the interviewee mentioned that, being a ‘DSL-expert’, he is often consulted only for DSL-based product-line projects.

“..and I mean just to add this one reason why I tend to use DSLs instead of feature models is simply because I am mostly called into projects that are of that nature because that is my focus...that's the one thing (means his main expertise is with DSLs). I mean, I think I do, obviously, I probably have a buyer to DSLs, but I'm aware enough of the other stuff (means feature modelling), so, I think that I can judge, but I don't, not that much involved with these (feature modelling) projects because other people see me as a DSL person.”

4.2.4 Case Study 4: Automotive Company

The fourth case study describes the product-line and variability modelling practices of an automotive company that manufactures both hardware and software for different car models. The company manufactures 400,000 different car models every year. Three main product-line platforms are maintained, from which all the car models are developed. The interviewee is a software architect who was previously involved in modelling and managing variability for the company’s product line. Currently, he works as a researcher in the company, and specializes in future strategies for developing software and software architecture.

A. Variability Modelling and Realization

Product-line adoption: - For this company, the transition to a product line approach was a slow evolutionary process was spread over 10–15 years, rather than a conscious decision. The product line was built by identifying commonalities in software used in different car models. The car models share

basic software units or subsystems, while differing in their own subsystem features. For example, multiple car models share the same engine-control system, but can differ in their safety features. The interviewee indicated that the company has always focused on ways to increase reuse by identifying commonalities in its products. Each subsystem is developed and maintained by a separate team, composed of software developers and architects. There are constant interactions among the development teams maintaining the different subsystems, given that a single feature, such as adaptive cruise control (ACC), is realized by several subsystems co-operating with each other.

“If you look at one of the most complicate features like the adaptive cruise control, as you experience it as a customer or driver, it is actually realized by 18 different subsystems, which is obviously designed by 18 different teams because you have the radar, engine, UI, the big graphical displays, you have the pedals which is a separate subsystem, and so on. So, in many cases, teams need to collaborate in order to realize the features... they need to agree on the interfaces, they need to deliver the software towards the integration point in time, and so on.”

Creation of the variability model: - This company uses feature modelling at three levels to specify variability: top level, intermediate level, and low level. At the top level (or complete vehicle level), there is one feature model, consisting mainly of customer-visible features. This feature model is built and maintained by a centralized team, and is used to reach agreement between the research and development departments on product planning and to understand the scope of features belonging to products. In addition, this feature model helps developers to trace the implementations of subsystems back to the respective features. This top-level feature model is a very basic tree, and the features are grouped according to product functionalities.

“At the very highest level, we have a feature definition of what is optional and standard (features) in the car, it’s more practice, now it is stored in the database, but 10 years ago actually it was in a Excel sheet... the features, there is no conscious meta-model of the features, it’s more, I would say, it has also evolved over the last 15 years, as a way to determine the content of a car, in the sense that we can agree what should be offered to the customer or not, should the car have rear-doors or child locking system, a remote control or not, automatic transmission or not... it is customer-visible features, but it has also customer invisible features, should it have remote-diagnostic stuff, and so on... At the top level, there is some structural (functional) grouping, like say chassis features, comfort features, power-train, and so on... The feature list has more than one purpose; the most important purpose is to agree between the R&D for product planning or organization and for knowing the contents of each product. Based on that, you derive the requirements on each subsystem to realize the features, in most cases, these features are realized by several subsystems co-operating...”

At the intermediate level, there is a single feature model; more information on the intermediate-level feature model is not currently available. At the lowest level, there is a separate feature model for each

subsystem, developed and maintained by the development team that owns that subsystem. The feature models are developed and maintained using TeamCenter [83]. Not much information is available on the process of creating feature models.

Modularization and structuring of feature models: - The top-level feature model consists of approximately 300–500 features, and the intermediate feature model consists of 300 features. The number of features in the low-level feature models varies among subsystems; the infotainment model has the highest number, with up to 3000 features. The interviewee said that it is hard to confirm the exact sizes of the low-level feature models because the numbers of features keep changing. The height of the low-level feature models varies between 2-3 levels. We do not know whether the interviewee refers to the top, intermediate, and lower levels as being different hierarchical levels of a single feature model, or if they actually exist as separate feature sub-modules. For the rest of this thesis, we will refer to these models separately as top-level feature model, intermediate-level feature model, and low-level feature models.

Feature dependencies, data types, and cross-tree relations: - Feature models in all of the three levels consist of only optional and *mandatory* features, and there is no feature-grouping. Since the feature models do not support grouping, and the features are maintained in three different databases (top-level, intermediate-level, and low-level), it is possible that the interviewee is simply referring to databases containing list of features rather than to actual feature models. Features are mostly of type Boolean. Cross-tree constraints are not explicitly modelled, in order to keep the models simple.

“No, we don’t do that (cross-tree constraints) a lot, we don’t do any explicit, what do you say, logical conditions for, no we don’t do that, it is basic modelling, no logical conditions between features, and so on. I would say, certainly there are conditions, but they are more implicitly captured in the requirements or documentation for the subsystem.”

Evolution of feature models: - This company performs a ‘change management update’ process twice every year, to manage changes to the feature models. The most frequent evolution types are feature additions and enhancements of existing features. The process of adding a new feature to a feature model depends on cultural and organizational constraints, as described below by the interviewee:

“I would say, the restrictions are more cultural or organizational because if you come up with a new feature, you need somebody responsible for defining the feature, may be writing the use cases or some other way for specifying the feature; you need to find organization for developing the feature or teams for developing the feature. If you look at the top level, the one with the 300 features at the complete system level, the turn-around time for actually incorporating an official feature (means feature list), that’s much longer when compared to the team that develops the sub-features that is contained within the vehicle sub-system. So, I would say, the constraints of incorporating a sub-feature in the feature model,

it's not so much dependant on technical aspects, as (compared to) the cultural or organizational process aspects."

Not much information is available regarding how the feature models and artifacts are kept consistent, as a result of evolution.

Mapping to artifacts: - Variability is modelled implicitly in certain design-level artifacts (including functional logic design blocks, components, and Simulink models), source-code artifacts (C, C++, and Java files), configuration files, and build files. A subset of UML is used to model the functional logic design blocks (that make up components) and their variations. Custom-made variability constructs and techniques are used in Vector-EC tool and Simulink to model variations in components and their behaviours, respectively. The developers use informal annotations to describe associations between artifacts, such as relating logical blocks to the features they realize or relating Simulink models to components. These annotations are inserted manually and are expressed using text or pictures; they are maintained in a proprietary database, or in a separate Word document, or as informal relations attached to design artifacts. These annotations are used mainly to understand the associations between artifacts, and between the design artifacts and feature model. Besides annotations, the developers have established a few traceability links among the logic blocks, Vector-EC components, and AUTOSAR components; more information on the type of trace links or traceability methods is not available. Currently, the developers are working on expressing trace relations between the Vector-EC tool and the feature model in TeamCenter.

The most commonly used variability realization mechanisms are #IFDEF directives, in source files, and configuration parameters, which are stored in a central configuration file. The configuration files are adapted based on a particular car configuration during start-up. There are a total of 270 configuration parameters, most of which are of enumerated or Boolean types. The interviewee indicated that not much information is available to him about the variability mechanisms used at the implementation level because most of his experiences were confined to the domain and architecture phases. The interviewee provided an elaborate discussion about the types of variabilities modelled in components and their relations to features based on presence conditions.

"In most cases, we don't have lot of different components realizing the same type of feature, the variability is more, if the feature in the car or not. For example, when you buy a car, as a customer, you say that, 'I want to have ACC', and that means you need to have lot more software compared to if you have just the regular cruise control; and that would mean that you will have more software components than in the second car (refers to the car with regular cruise control), and that would mean there would be some undedicated ECUs not used in the second car (refers to the car with regular cruise control). So, typically, if the feature is present or not.... We don't redeploy software components, and we keep the number of software variants as low as possible, and use some other configuration mechanisms. For

example, adaptation during start-up, you read a configuration file when you start the system, the configuration file is stored inside the car, and the configuration file is read every time you start the car...”

In the manufacturing department, the engineers use a proprietary product configurator called KDP, which was developed 20 years ago by IBM. KDP uses a tree-based model to configure ‘software articles’ for different parts of the car using feature selections. We do not know what the interviewee means by ‘software articles’; it is possible that he might be referring to various software or hardware specifications for different parts of a car based on a specific car model. KDP is also used to configure and maintain different software configurations for different car models. Variability is bound both at compile-time and at run-time.

B. Benefits and Challenges

We asked the interviewee about his neutral response to our survey question on the value of feature modelling (refer to question 1 in chapter 3). According to him, feature modelling, in its simplest form, helps developers to understand more about the scope of product functionalities, while does little to assist with product configuration and derivation activities.

“We build 400,000 products a year, if you look at the single-car model, if you look at the different software configurations that we can build; it exceeds more than 3 million. So, we don’t even build all the configurations (using feature models) that are possible, like I said, the car can have with or without automatic transmission . . . we can build with and without lot of things..., then if we then compare this with number of configurations that other companies have, I would say, we are dealing problem with order of magnitude or several order of magnitudes, more configurations than most other companies have, and we do that on a daily basis. On the other hand, in our development, we don’t do a lot of explicit feature modelling like grouping or inheritance or selection or...even automated tests if this feature construct is valid or not, like I said, yes, we are able to handle great number of feature variants and configurations with very basic means of (feature) modelling”

The interviewee expressed his concern in handling large numbers of dependencies between subsystems, and in establishing a harmonized collaboration between different development teams.

“We have a really complex system, in order of 100 different development teams, interfaces, several thousands of attributes/services, and many configurations... I mean, we have a lot of dependencies between subsystems and between teams, it is quite difficult for the teams to work autonomously... If we get to look at the present approach when it comes to modelling, it seems like we are aiming to keeping practices, which means that we are trying to align the modelling efforts between different domains, we try to align the design artifacts that we are using, and it is also focus on keeping traceability between the

different types of artifacts like I said the feature models, the component models, AUTOSAR component models, our architecture models, and so on, so, put a lot of effort in maintaining all these design artifacts in a consistent way. My personal opinion is, I don't think that is the right way to go because since the complexity of our systems is exponentially increasing basically regardless of what number you are looking at, we actually need to identify ways of working, the different development teams exactly can work more autonomously, that they can use tool that they need for their specific problems more independent of each other, and so on."

The interviewee described another challenge concerning variability at the code level: how the use of lots of #IFDEFs tends to compromise code readability.

"I have seen problems when using #IFDEFs in a lot if messy manner to be very precise; depends mainly on who writes the code, don't want to mention any suppliers."

4.2.5 Case Study 5: BigLever GEARS

The interview subject for this case study is a tool builder, BigLever, which has built a tool framework, GEARS, for systematically modelling and configuring variability in software product lines. The development of GEARS started seven years ago and its main focus is to support the automatic generation of products for product-line systems. Currently, the framework uses three components for automatic product generation: (a) a feature-modelling component, which models the features in a product line, (b) a set of configurable assets, which constitute various artifacts (e.g., requirements, source-code files) shared across a product line, and (c) a product configurator, which configures product-line assets to derive products. Until now, this tool builder has created product-line applications for many of its clients including army training systems, naval systems, patient management systems, airport baggage management systems, automotive systems, and web- and storage-server management systems. The interviewee holds the CEO position in this company and has played a major role in the construction of the tool framework. In addition, he acts as a consultant who teaches and advises many of the company's clients on variability modelling and configuration.

A. Variability Modelling and Realization

Creation of the variability model: - GEARS uses feature modelling to construct variability models. Because the interviewee played a major role in guiding most of the company's clients in building their feature models, we asked him to describe the processes used to introduce the framework to clients and to construct their feature models.

According to the interviewee, most of their client companies are divided into teams. In most cases, a company employs roughly a hundred developers who are organized into teams of roughly size 10. Each

team owns a sub system. In most of the companies, the chief architect or the lead product-line engineer is responsible for the overall structure of the sub systems and the overall feature model(s). For each individual sub system, the corresponding team leader or architect or lead designer creates and maintains the sub system's feature model, because he or she possesses detailed domain knowledge regarding the sub system. The interviewee mentioned two exceptional cases that had more than a hundred developers and sub systems. For example, one case involves an automotive manufacturer that develops around three hundred sub systems and employs around 3000 engineers. According to the interviewee, these cases challenge the scalability of variability modelling and management. In both cases, there is an interest in organizing a centralized team dedicated to the overall management of feature models.

The interviewee states that there are two common approaches adopted when introducing the tool framework for constructing feature models: a top-down approach and a bottom-up approach. In the top-down approach, the feature model is based on the high-level product differences or feature differences that are visible in the given product specifications. The top-down approach is usually used when teaching a client company's lead requirements engineer or a lead architect to build feature models. BigLever conducts a mini 3-day pilot project to train the company's clients in feature modelling and to familiarize them with the tool framework. The clients are walked through a procedure to find, extract, and capture features. During this exercise, the client company's modeller or designer or engineer is repeatedly asked: *"What is it that causes one of your products to be different from another; think about the different flavors you deploy and start to tell me why some of them are different than the others?"*

According to the interviewee, the modeller or engineer usually starts answering the above question by pointing to entities from the lowest abstraction level (the implementation level). Subsequently, the interviewee continues asking "why" until the engineer starts thinking in terms of abstract features. Once the answers are obtained, the interviewee teaches them to determine the type of features, in terms of the basic types available in the tool framework:

"In our language, we have types that are familiar, what we call enumeration types or Boolean types for features. You create a feature tree that looks like very traditional feature tree, like in terms of defining a feature where you can choose one and only one value, we call that enumeration and we put radio buttons on the graphical visual, so it's clear to them. This is like define a set, you teach them how to do that, you teach them how to think in terms of the set type which is, if multiple selections use the checkbox in the list..., if Boolean then it is a single checkbox. So by doing this, we sort of guide them by usually saying, use an enumeration for what you just told me or use a set type, and basically get them to capture 1 or 2 of those."

By the end of this initial feature-capturing process, the interviewee encourages the modeller to return to the flip chart and draw an initial feature model that captures the identified features in their domain, and

guides them to subsequently update the model to include only the appropriate feature choices or decisions. Finally, a small initial GEARS feature-model prototype that matches the client's requirements is constructed. This process of feature-model construction using the top-down approach is described below by the interviewee:

"You know, if you are doing management systems for a hybrid vehicle that keeps the batteries and electronics cool as they are charging and discharging, then they are drawing pictures of, you know, fluid flowings and things like that, but they'll start to then describe where in that picture the things are very different on one situation and in another. As soon as they start that thing, then, you know, you are pushing them to capture those decisions or those feature choices into the feature model. So it's a bit of a guided exercise or fun; you sort of to teach them how to think, how to ask, how to draw to extract those features. It tends to be very painful for the first day or so to get your feet on solid ground, but by the second day, you know, they grab the feature model that seems to be pretty solid, and then you get them to start instantiating, like to think about one of the systems that you built, then go through this decision model or feature tree and make the decisions and see if that is a natural way of representing the way you think about your instances. If they come across something as, 'nah.. this is really...', they'll typically find these soft spots in the feature model, and then go and refine it from there, but it's usually the validation of have they captured a good feature model is: can they walk through the process and say yeah, this is how I think about creating a new product or extend the product line. So once they can do that, then it's like, we captured the thought process of building their systems."

The interview refers to the second approach as a bottom-up approach, in which the feature model is built from existing product-line artifacts, especially from source-code implementations, build files, and configuration files of products. In order to identify and extract features, the UNIX diff operator is applied to code clones to obtain product differences; each time a difference is identified, the reason for that specific code variation is analyzed:

"So you take a module or you take a subsystem and then you just run a UNIX diff, and every place you find something different, we go back to that technique and go why, why is there something different here?; could be arbitrary variation is induced, somebody cloned a copy of the system and modified something just because they want to make it better, (interviewee quotes clients: 'Yeah we modified the system because blah, blah, blah...'), and that's where you start the process of, why did you implement that different, and then why why why why why, until you get to the essential difference. So, now you got variation point you can put in...."

Each variation identified by the reasoning process described above is recorded as a variation point in the code artifact, and a corresponding feature is added into the tool's feature-model space. Every time a particular feature for a product is selected in the feature model, the corresponding variation point in the

code artifact is included in the product's solution space. The advantage of applying bottom-up approach is that it helps to map features to artifacts:

"...So, you do both the discovery of where things are different, you introduce the variation point in the mapping and you capture the feature, all at the same time."

Once an initial feature model is obtained, it is validated by selecting different feature configurations and generating the corresponding product instance within the tool. This process checks that the feature-model structure is acceptable and that each generated instance possesses the expected functionalities for the derived product. On the contrary, if a generated instance has shortcomings, a secondary refactoring process is performed:

"When you are done with that process (initial feature-model creation process), you might, now see where the variation is and how they are implemented, encapsulated. In case they (clients) don't like it, I can eliminate variation points or make it smaller or bigger or whatever. That's a secondary process of making it better, typically the first thing we'll do is combine all the 2 or 3 copies of code of products into one, now that's going in the opposite direction with our generator, make our feature decisions, and push the button, and GEARS will output a system. Now you run the diff with the original, and convince yourself that you have implemented your variation points with features correctly."

Modularization and structuring of feature models: - Most of their clients' feature models consist of fewer than a thousand features. The interviewee did not specify an exact number or range for the size of the feature models; however, he mentioned a naval system's feature model having 200 features. The heights of the feature models do not exceed more than three levels. The interviewee makes an interesting remark that suggests that systems, in which the number of features is in the magnitude of thousands, are extremely rare.

"We haven't seen the thousands of features yet that I hear about, other people talking about, when I hear those big numbers, I wonder why is there so much variability? Customers don't think in terms of that much diversity in the products. So I am wondering if it's because things haven't been abstracted high enough, you end up with about 1000 or 10,000 features."

We asked about the process used to structure, name, and document features within feature models. The interviewee encourages modellers to have long feature names and to avoid the use of acronyms. The purpose of this guideline is to make it easier to understand the feature model and its contents. For each feature, the tool offers a description field that provides a rationale for selecting the feature. The interviewee claims that the description field proves quite beneficial for extending the knowledge of features to business and technical people, who might have limited knowledge of product features and who are occasionally required to work on the feature models. When we asked how the right granularity or abstraction level for defining features is determined, the interviewee responded that there is no clear-

cut method or heuristic used. However, for the majority of clients, the reasoning processes used in the top-down and bottom-up approaches often help the interviewee in determining the right level of abstraction for features. In one or two cases, there was a need to include in the feature model very technical features, such as certain fine-grained debugging features (details not available) for a server product line developed for an online vacation-rental marketing system.

We asked about how teams reach consensus on overlapping feature concepts and inconsistent terminology used across multiple feature models that are maintained by multiple teams. The interviewee described a new capability within the GEARS framework called a *mixin*, which is the extracted set of features that can be shared across two subsystems. If a feature overlap occurs between sub-systems, a *mixin* is created such that features are imported into the mixin from the overlapping sub systems by eliminating duplicate features and feature conflicts, if any. However, the interviewee indicates that it is difficult to avoid feature conflicts and inconsistent representations of the same feature in cases that involve lots of feature models. In these cases, most of the teams tend to focus only on their own feature models because they do not have a detailed understanding of the features that are spread across multiple sub systems.

We asked if multiple feature-model views are requested by clients, and if this capability is supported within the framework. The interviewee described three feature model views that are offered by the tool: (a) a plain-text view with feature names, types, and description field; (b) a structured-textual view with drop-down menus for selecting predefined data types for features; and (c) a graphical view which visually depicts the hierarchical structure of the feature model. The interviewee said that a few clients have expressed interest in additional feature views that could be useful to non-technical users, such as managers or business people.

Feature dependencies, data types, and cross-cutting relations: - To support feature dependencies and groupings, the GEARS framework supports four basic types of feature and feature aggregations: (a) an enumeration type in which exactly one sub-feature is chosen from a set of sub-features owned by a feature; (b) a set type in which zero or more features are chosen from a set of sub-feature choices owned by a feature; (c) a record type in which all sub-features of a feature are chosen; and (d) a Boolean type which applies to a single feature, in which the feature is either selected or left unselected. Each feature can be of integer, Boolean, float, or string type. Default values for features can be specified for the most commonly used features, leaving less-used features undefined. Cross-cutting relations between features are supported in the form of assertions, which are expressed using propositional logic formulae. The interviewee says that modellers tend to limit their use of assertions to those unexpected situations or bad feature combinations that are to be avoided (specific details regarding such situations are unavailable):

“The use of constraints, we call them assertions, tend to be pretty limited..., so, it’s for whatever reason people put things in to protect for the really awful things from happening. But usually, what they are doing is, they create a instance of your feature model and that becomes a persistent thing that gets used over and over and over again during the development.. Assertions to avoid the real bad stuff are: when they come across unexpected problems, they’ll put that into assertions to see that, that never happens again.”

Configuring feature models: - GEARS uses feature profiles to manage large numbers of feature combinations among their clients’ feature models. The lead designer for each sub-system defines a feature profile by restricting the features choices allowed within that sub-system feature model; these feature choices are available to other teams. To derive a new product instance, the application engineer selects sets of feature profiles. The interviewee emphasizes that this feature-profiling capability of the tool helps the client companies to reduce and manage an explosive number of feature combinations within their sub-system feature models.

“You might have 10 or 20 features in your subsystem, which, if you do the multiplication it could be huge, billions, trillions number of atoms in the universe, whatever. But, those people (refers to the lead designers from client companies) decide how many combinations to expose to the next level up, so through defining feature profiles they say: ‘out of all those possible combinations, 23 of these are interesting; those are the ones that we will test, that we will build, and will guarantee that will work’. So now, instead of having billions of possible combinations of 20 features, I just exposed a linear list of 23 off the shelf assemblies of that component... So the profile is, I instantiate my feature model with a set of those choices, I give that sort of choices a name, and that’s the profile. So now, somebody at the next highest level gets to say: I can choose this profile, this profile, but I’ve never seen the internal features (refers to features within sub system-feature models).”

Evolution of the feature models: - The interview included a few questions on the stability of feature models and the types and frequency of changes that occur in feature models. For the majority of clients, feature models evolve frequently, mostly because of feature additions and refactorings. In most cases, these changes do not significantly affect the structure of the feature model, except in large-scale systems (e.g., the automotive manufacturer mentioned earlier whose product line comprises more than 300 sub systems). In this automotive case, a large number of refactorings compels the modellers to frequently reorganize their feature sets either by renaming features or by using alternative ways for ordering features in the feature-model hierarchy. When the addition of a new feature affects other features or the existing structure of the feature model, the interviewee encourages the modellers to first add the single feature to the feature model, and then update the related profile(s) and variation point(s).

Mapping to artifacts: - Variation points are used within the framework to map features to product-line artifacts (especially source-code artifacts). Each variation point specifies a mapping between a set of one or more features and the corresponding product-line artifacts, expressed using a specialized logical language.

The client companies use a variety of mechanisms to realize variability within product-line artifacts, including #IFDEFs, inheritance, annotations, build flags, and configuration-file settings. The interviewee recalls configuration files being the most commonly used and #IFDEFs being the least preferred:

“It’s been bad enough... Nobody will ever use the #IFDEFs...No one actually uses #IFDEFs, so, we see runtime configuration, with configuration files it is pretty common. “

Interviewer: *“so you configure or generate those configuration files or..?”*

“That’s the first place to start (refers to the generation of configuration files), you automatically fill in values in configuration file, the problem with that is, now you have no... you don’t have explicit boundaries around your variation points, if something in the code and I can’t tell the difference between product-line variability and normal conditionals within that code. So, we encourage people to go in and take those configuration settings that makes sense, and then replacing the code that refers to a runtime configuration, turn it into a static explicit piece of variation point, now that I can scan my code and see where that product line variation point exists, you know, querying GEARS and show me all the places that feature X actually impacts my system if you have a configuration file only, then that shows up is the entry in the configuration file.”

One interview question focused on whether and how the client companies derive test cases from feature-model configurations. Most of the client companies manage their test cases the same way that they manage other product-line assets that have variation:

“..basically if I can pull out a functionality in my implementation by turning feature off, then simultaneously want to pull out the test cases, that test that capability, so you’ve implemented the test cases exactly the same way as you do in your code or in your requirements.”

B. Benefits and Challenges

There are a few interesting remarks from the interviewee concerning the advantages of feature modelling, the adoption of a product-line approach, the use of their variability-tool framework, as well as the feature extraction and modelling techniques used to train the client companies’ modellers or designers.

The interviewee described how his company’s 3-day training project helps modellers to employ both top-down and bottom-up feature-model construction approaches for extracting and modelling features. This

project helped to shift the thought processes of the modellers from being product based to being product-line based, where the modellers consider features to cross-cut across all products.

“If we send them (clients) a copy of GEARS and a copy of the user’s guide, and we give them the WebEx, they just do awful things, just always a failure because people don’t, they never learn how to think this way. But, we found that if you can get them through the 3-day training class... We call it the SPL epiphany and this thing just goes off and you see the little white bulb, you know, go on in their heads, and I don’t know exactly what the deal is, but they get this realization of getting away from features, from product, separate ways of thinking, and they just tilt their head 90 degrees and see the world like features that cut across all the products. And as soon as that happens, and you have taught them enough good techniques on those 3 days to capture a feature model, and you leave them with something we feel comfortable with, and then they usually continue and create something that’s pretty good.”

The interviewee observed that developers and engineers are interested in using variability-management techniques, such as refactoring, because these techniques can reduce their development effort and the size of their code artifacts considerably:

“By our measure is how efficient people can get their work done...So, we observed this really interesting behaviour that natural human laziness will cause good things to happen. People refactor their variation points so that they can go home 5’o clock instead 6’o clock, then anything that makes their life easier in a product line setting, they’re refactoring, they’re reducing, they are making better use of their variation management techniques, so you observe this in a couple of places that made over the course of the first 2 or 3 years. You actually have the total lines of code in the system just constantly going down, and they drop by 30% or more over the course of 2 years; even though you are adding more features and more products and more variants, your code’s shrinking. It’s just that human laziness is making a really good thing happen. So that’s really surprising, that sort of suggested something really good about the way you were making people work.”

4.2.6 Case Study 6: pure::variants from pure systems

Similar to case study 5, this case study involves a tool builder that has seven years of experience developing product-line management technologies for modelling and configuring variability in software product lines. Their main focus is on improving the reusability of product-line assets. Their product, pure::variants, supports variability management at different stages in product-line development, including requirements, design, architecture, and implementation. pure::variants uses feature modelling to model and configure variability. New extensions are currently being added to integrate this tool with other modelling environments, such as AUTOSAR and UML-based models. We interviewed 2 employees: one who holds a director position and another who holds a managing director position with the company. Both interviewees also consult on product-line projects, especially for embedded systems. Both interviewees are actively involved in the promotion of pure::variants and have played major roles in

the design and development of pure::variants. In order to separate the interviewees' viewpoints and avoid loss of information, we describe each interview separately. We conclude with a comparison between the interviews. The first interview was conducted and transcribed in German and then translated into English, and the second interview was conducted in English

4.2.6.1 Interview 1

This interviewee has more than eight years of experience in variability modelling, especially on embedded-system projects. The practices described below are based on the interviewee's experiences, mainly with industrial projects involving the company's clients.

A. Variability Modelling and Realization

Creation of variability model: - The interviewee described two common approaches to construct feature models. The first is similar to the top-down approach described by the GEARS interviewee, in which the features are identified from requirements and the feature model is constructed. According to the interviewee, this method is usually used by those client companies that prefer to keep their feature model(s) separate from implementation-specific details – such as original equipment manufacturers (OEMs) (e.g. automotive companies) that have a clearly defined product structure and requirements. Depending on the client company's interests, the feature model(s) may or may not be configured to derive products in the later development phases.

The second approach is similar to the bottom-up approach described by the GEARS interviewee, where a feature model(s) is constructed by extracting features from existing sources, such as any form of existing internal variability models (e.g., an Excel spreadsheet or a database of parameters), or existing product-line artifacts. In the first case, the customer has some form of internal variability models already built; these models often have fine-grained variability elements (e.g., parameters) that can be analyzed to identify features, with which one can construct the feature model. In the second case, experts analyze product-line artifacts (especially, source-code artifacts) to identify features based on product differences; analyses include comparing the status of #IFDEF variables, code branches, or code versions associated with different products.

“For many (clients) it is basically an internal variability modelling in some form: Excel spreadsheet and so on. They have, so to say, such things exist, those who already had a certain idea, which they extended, which is more or less compatible, and which they work up (means to process, improve, to work with for improvement). That is one of the sources, which basically, the granularity there is finer than what a feature is, because it's often parameter-oriented and represents a much more fine-grained level of detail than what a usual feature model is. It's about combining, copying, structuring. That's one of the kinds (of customers). These are those who are relatively advanced already. And then there're those, where it's

source-code-based. That is, there, the source code, in multiple forms (representations), is analyzed and worked up. Basically you know how, #IFDEF is, so to say, the one classical form... likely in combination with branches and textual comparisons. That is the embedded world. Those who basically have put everything in code without having any real formalization... there's typically some more work (e.g. refactoring) necessary, since the differences in source code are no features."

When asked whether the top-down or the bottom-up approaches are used most often and are most successful, the interviewee responded that the approach used depends on the client company's interests. The interviewee presented an interesting comparison between the two approaches, based on who within the client company handles the process of constructing the feature model. The bottom-up approach is more successful with developers and technical people, whereas the top-down approach is preferred by domain experts. The two approaches result in very different feature models of the same system.

"It is often, (clients) tried to do both, and one or the other is more successful. When one lets the technicians (technical guys) do it, then the development from bottom to top is more successful. In turn, letting developers do the derivation from the requirements is rather difficult. And in turn, the same happens when you let the domain requirements experts do something like that. Then, they of course describe it from their view. Thus, you get very different models when they basically work on the same system. Respectively, one group is not able to directly derive it from the other's input. That was also an experience at least in one project, where the others (technical guys) were not able to create useful feature models on the basis of the requirements on which they develop the systems, but rather from looking at their code."

Modularization and structuring of feature models: - Feature models constructed using the top-down approach consist of relatively abstract features and typically have fewer than a hundred features. In contrast, feature models built using the bottom-up approach have relatively technical or fine-grained features and the number of features can be on the order of hundreds of features (typically between 500-1000). According to the interviewee, the smallest feature model that went into production had 50-60 features; it is not clear if this feature model was created using the top-down or the bottom-up approach. The largest feature model had 5000 features; however, it did not go into production due to restructuring and timing constraints (details are not available regarding these constraints). Feature models have an average height of 3-5 levels, with the exception of one case, which had more than 10 levels; not many details are available regarding this specific case. The interviewee reported that the feature models are usually very flat. In the interviewee's opinion, flat feature models are less desirable. One reason could be that such feature models require the need to introduce additional cross-tree constraints, which might lead to more complex models. The feature models include mostly functional features, although quality (non functional) features that have a direct impact on functional features are also sometimes modelled. As a

side note, the interviewee suggests that there is no relevant distinction between functional and quality features.

“This discussion comes up often, but on the feature level, there's no difference. For me, it's a completely artificial differentiation between so-to-say non-functional and functional features. Why is safety, so-to-say, a non-functional requirement? I mean, it has clear impact on the architecture and has to be implemented. I disagree a bit here. The fact that something has influence on architecture doesn't make it a functional requirement. That is, non-functional requirements, such as performance, maintainability, etc. always influence the architecture... Performance as well, when it has a technical impact. I mean, you can say everywhere "I have a feature that expresses that the system is fast", but it doesn't have a connection to it, then you don't need it, of course. But usually, you can decide, e.g. through the hardware that you employ, etc. It's only about representing variations, and not when a non-functional feature requires variation in the realization.”

Although pure::variants supports modularization of feature models, the interviewee says that most clients construct a single feature model. The only exception is the project mentioned earlier that had 5000 features. In this case, there was a single top-level feature model (or super model) and 50 smaller sub-models. The super model represented a course-grained view of the domain and modelled mainly commonalities among the products; the number of features was in the order of hundreds. The sub-models typically had fewer than a hundred features each. The sub-models had references to the super model, but not vice-versa. A hierarchical naming schema based on path names is used to name features; that is, each feature had a long name, which combined the names of its ancestors. This naming schema helped the developers to avoid naming inconsistencies or collisions between features. The interviewee mentioned that certain features were shared between the sub-models.

Feature dependencies, data types, and cross-cutting relations: - Among feature-dependency relations, alternative groups are the most commonly used, where *XOR* groups are used more frequently than *OR* groups. Boolean is the most commonly used data type for features. Features sometimes have enumerated attributes attached to them. The tool provides a *DefaultSelect* mechanism that allows users to specify default values of features. Unfortunately, this mechanism is less frequently used, and we could not get more details about the mechanism from the interviewee.

Their clients' feature models include very few cross-tree constraints, with the most common constraints being *requires* and *conflicts*. However, the interviewee says that more constraints are used in the family model to restrict combinations of the same product. These constraints are simple Boolean expressions or conditional expressions joined by *AND/OR* relations. In addition, a feature model can contain soft constraints (or weak relations), such as *recommends*, that convey the preference for choosing one feature over another. The interviewee suggests that these weak relations help the *DefaultSelect* mechanism to select the best available default values for features when a feature configuration is partial.

Mapping to artifacts: - A family model is used to model product-line artifacts and to map artifacts to feature model(s). The most common product-line artifacts are source-code files (mainly C/C++ and occasionally Java files) and configuration files. Feature models are mapped to the artifacts using simple Boolean constraints. There are two representations used for the family model: (a) a single model (otherwise called a *base model*) that presents an overview of commonalities and variabilities in the artifacts, and (b) annotations within the artifacts that express variabilities (usually in the form of variation points). In the project that had 5000 features, the interviewee reported that every feature sub-model was associated with a family model. The super-model had no corresponding family model.

Although the interviewee indicates that many mechanisms are being used to realize variability in product-line artifacts, #IFDEFs are the most common. He says that developers often had bad experiences using #IFDEFs, and hence #IFDEFs are less desirable. In some cases, alternatives to #IFDEFs are used – such as by deferring the binding time to runtime in configuration files (IFDEFs supporting static binding time only), or by adding #includes.

“Well, basically everything is used, I mean, everything what is available is used. Well, those mechanisms with which the developers made good experiences. #IFDEFs, yes, of course they appear.... #IFDEFs are of course well spread, yes. But, we now have customers that limited themselves in the implementation, and therefore were able to forego #IFDEFs, in exchange for the possibility that they can decide over the binding time. This basically means that they cannot use a #IFDEF anymore, because there it's static, and that fits for them...”

Although configuration files are pre-generated (statically), a majority of clients currently prefer to use these files dynamically. The interviewee had a couple of discussions with client companies regarding the customization of configuration files. For example, a few clients expressed interest in abstracting configuration parameters away from technical details, allowing them to alter the parameters as required.

“We already had discussions, how can I abstract from that (configuration parameters), when it's bound and how we, so-to-say, get a certain level of abstraction from the technical realization and provide a uniform view. But, I cannot see any tool that currently has the optimal solution. But, that is what will come. I mean, when it's rather model-based, then there's not a big difference. When one has the model-based thinking, then there's no difference how it's represented. You only see the parameter or the setting or the variation points. The mechanisms are more hidden, than with a #IFDEF, where it's quite explicit. Basically, that's what interest people.”

4.2.6.2 Interview 2

This interviewee works on improving the architecture of the tool chain. In addition to consulting and tool promotion, he assists with adding new capabilities to the tool chain. He is involved in both academic and

industrial product-line projects, which mainly include automotive systems, server-based systems, and wind-power measurement systems.

A. Variability Modelling and Realization

Creation of the variability model: - This interviewee mentioned using both the top-down and bottom-up approaches described by the first interviewee to construct their clients' feature models. He indicated that the top-down approach is used when consulting with modellers or designers from client companies.

"We have some idea about architecture, about product, or possibilities, which product should have, and then we have some feature in mind what we can create, quite easy to fit this feature that feature into the model, and to say.. something should be in every product and... something is special to the customer, or we have some features which all products have. But it's mostly structural..."

The interviewee described an additional source to build feature models for the bottom-up approach described by the first interviewee, despite existing artifacts: a feature model can sometimes be constructed from existing internal models (termed '*solution models*' by the interviewee) that have variabilities (e.g., a database containing variable parameters). He mentioned that these internal solution models can sometimes be augmented with information from the artifacts; for instance, a new parameter can be added to the parameter database based on a corresponding #IFDEF variable. The interviewee helps modellers to build feature model(s) by capturing features from the artifacts or solution models. In some cases, specialists manually mark differences in the code fragments belonging to different products. For each marked difference, the developers then decide if a new feature, sub-feature, or a new attribute is to be created in the feature model. Once the feature model(s) is constructed using pure::variants, the interviewee helps the modellers to associate the feature model(s) with the artifacts or the solution models in the pure::variants configuration space.

Modularization and structuring of feature models: - The interviewee indicates that features are usually of a technical nature, especially when the bottom-up approach is used where features are identified based on technical differences that exist between the products. The interviewee points out that each feature or a set of related features should represent some coherent unit of functionality.

"There should be clear functionality which should be expressed as a feature, or by the feature tree; I would say, because there can be sub-features or options. But, there should be a clear functionality. I think, sometimes we see features like CPU types for instance, so, in my opinion it is not a good feature. I would say, it (CPU type) must be something like an attribute (with different values), because it makes different behaviour., it is not really a feature to plan on..., there's something different, but it is not a feature."

According to the interviewee, the sizes of feature models have been manageable for all projects until now. The number of features in a feature model is on the order of hundreds for both industrial and academic projects. As an example, the interviewee mentions an academic braking-system project with 180 features and a current industrial project (consisting of 12 products) with 120 features. There is no mention of any cases where modularization is being applied in industrial or academic projects. However, the interviewee describes the concept of configuration space in pure::variants, which is used to modularize a feature model into sub-models. In addition, the tool allows a user to rank the sub-models to specify the order in which sub-models should be configured for deriving products (similar to staged configuration).

“We have so called configuration spaces, where we can find which feature models work together, you can have any number of feature models in your project and then in configuration space; you say this model, this model, this model, are needed for configuration and you can also give them rank which is something like hierarchical, say: this is most top feature model thus needs to be evaluated first, and this is evaluated second, and so on. So, you can verify results from the top model, next level, and so you can see models which push the decisions down to the system; you can exactly say where some architectural model pushes some decisions into the functional model or something like that... We have very loose coupling here; when we write feature models, we can verify features, any feature you like in your rules; so, you get warnings if feature cannot be found, so we look into the project and we search for models... then we can combine the features. This is some kind of reusing feature models: you can combine differences by using configuration spaces...”

We asked how feature terminology is kept consistent in projects with multiple business units or teams. In most cases, modelling teams agree among themselves, and the interviewee helps the modelling teams to integrate their respective features and their relations in an integrated feature model.

The interviewee indicates that pure::variants allows users to set permissions on feature models, which is a capability used mainly for server-based projects. For instance, one can set read permissions on certain features, such that only specific users or development groups are able to view those features. It is also possible for a user to set configuration permissions that restrict users' feature selections. The effect is similar to staged configuration, where a feature model is partially configured based on pre-defined constraints.

Feature dependencies, data types, and cross-cutting relations: - The most frequently used feature dependencies are alternative groups, especially *OR* groups, and *optional* features. The most commonly used data type for features is Boolean. Attributes can be attached to features, as indicated by the first pure::variants interviewee. Each feature has one of three states during configuration: (a) *selected*, which means that the feature is included in the configuration, (b) *excluded*, which means that the feature is not

included in the configuration, and (c) *unselected*, which means that no decision has been made about whether to include the feature. During configuration, default values are automatically set for unselected features by the tool's validation engine.

Cross-tree constraints between features include mainly 1:1 relations, such as *requires*, *excludes*, and *recommends*, and 1:n relations, which are specified using Boolean expressions. According to the interviewee, soft constraints, such as the *recommends* relation, are useful for choosing default values for unselected features during configuration. The interviewee indicates that the tool uses a simple constraint language based on Prolog that allows users to write Boolean expressions and simple *AND/OR* relations over Boolean expressions. However, there are a few cases in which customers request complex rules, such as conditions to ensure that the maximum size of memory will not be exceeded. In these cases, the interviewee mentions that his company offers certain custom-specific functions in Prolog, or provides an extension to the Prolog constraint language within the tool chain (details on these complex-rule handling functionalities of the tool are not available).

Mapping to artifacts: - The tool uses the concept of a configuration space to model product-line artifacts and to map artifacts to the feature model(s). The product-line artifacts are modelled using family models, otherwise known as solution models. The main product-line artifacts supported are source-code artifacts, which mainly include C/C++ files, build scripts and makefiles. Both static and dynamic configurations are allowed within the tool's configuration space.

The interviewee presents an elaborated discussion on cross-tree constraints (rules) that exist (a) between features, (b) between family models in the tool's configuration space, and (c) between feature models and family models. He indicates that the family (solution) models tend to be very large and difficult to manage. The tool provides an auto-resolver that resolves conflicts among constraints during configuration. The auto-resolver presents to the users warnings of potential feature conflicts and assists users with possible choices for resolving the conflict(s). If the user proceeds with configuration without resolving conflicts, the auto-resolver automatically will select the best available solution, based on a set of internally-coded BDD rules and the nature of the conflict(s):

"If we have error, we get normal solutions. We have an auto hand-coded auto-resolver part, which takes the errors from the validation engine and tries to find some classified faults, and possible solution, select them, and again tries to use some internal rules to find some (better) solution for this. We are also able to auto-select stuff, if we have errors; so, you get mostly the best model we know. We can also process all errors, using BDD... it's programming to get all the list of errors which the validation engine returns, and then try to find solution for some known cases. It (engine) tries as long as some maximal limit of tries, and if we find no more changes in the relationship's conflict, there's nothing to change anymore, then it stops, and gives us the validation. So, you get always minimal errors to user back, everything can

be automatically calculated. We can also resolve, if user makes some mistakes; at the same time, you can also disable the auto-resolver in the tool.”

The same Prolog constraint language is used for writing constraints between the features within a feature model, between artifact elements in the family models, and between feature models and family models. The interviewee usually advises his customers to include simple Boolean rules in the feature model, and to include implementation-specific rules in the family models. The interviewee says that certain constraints in a feature model can be implicitly deduced from constraints or rules on artifacts in the family model, based on implication; this helps to reduce the number of cross-tree constraints in feature models.

In most cases, a large number of rules are specified in family models. For instance, the interviewee mentions a total of 130,000 rules defined for the family models in his current project:

“...130,000 rules or something like this on the solution, this means there are 130,000 differences... We have rules based on differences in the solution model, there are attributes that are used for feature references... Take one product, bring this to the model, and then you take another one and merge this into the same model, and everywhere you find differences merge elements, or say there wasn't elements that just was not there before; in this case, then we create a provision for this element... I would say, may be maximum 5% or 7% (refers to the percentage of constraints), because if you also model source code patches, lot of files exist already for compiling..., we model things like header files which defines variables, we have more rules because we only model variables, they are mostly changeable by features and then you have every element... so, we have lot more rules in the solution models than we have (in the feature model)... That is, every time you see element (attribute), where you say you have attribute name a,b,c,d,e, and then we have restrictions on their combination of values, so, you get something like 204 restrictions with these attributes...”

B. Interesting Remarks and Recommendations

Interestingly, the interviewee reports that the modellers and engineers in client companies often experience difficulty switching their mindset from a product-based scenario to a product-line scenario, and to think in terms of features:

“There were teams working before it ... there was a lot of existing code which needed to be transferred into something that can be used by product line, and until then, you cannot start the featurization (to identify features and build a feature model). We have to first put this code together in some form, and then it is easier and also for the people for beginning to think in terms of the products. It is one of the biggest problems in some companies: they come and develop products, they think in products and not product lines, it is quite hard for them to switch, to implement features, they implement products.”

In another interesting remark, the interviewee recommends that feature models be domain-specific to improve the communication among various product-line development teams and improve the understanding and configuration of features within a specific domain.

“...because you can model feature model, independent from solution models, you cannot always say you can change the feature models, but you happen to make solutions because we don’t know all the models which are needed for a configuration... This was a decision to make easy means of feature models. I think you know, for Windows, Linux, ‘maker of features’, if you have something to do with software, you can mostly see the feature model which contains operating system features, and why is that there is no operating system feature model, which is always useful. If such a model exists, then you can simply integrate it into your configuration and the features can be verified. And so, if you talk about the Windows feature, you need exactly this feature from this operating system or that feature..., users would love it... I think it’s something which you need sometime... standardized feature models for standardized problems...something like AUTOSAR, more easy to provide information and configure system.”

4.2.6.1 Comparison of Results from Interview 1 and Interview 2

In this section, we compare the responses of the two interviewees from pure::variants. This helps us to obtain a complete picture of the variability-modelling practices adopted by the company’s clients, and to cross-verify the feature-model data, such as size of feature models, types of dependencies and cross-tree constraints, modularization of feature models, feature data types, and so on.

Both interviewees hold top positions in this company, and each possesses more than seven years of experience in variability modelling, especially with embedded-system projects. They both have participated extensively in the design and development of the company’s tool chain, and they work actively to spread and promote the use of the tool chain. Table 5 presents a comparison of the interviewees’ responses about feature models. We found that there are a lot of similarities between the interviewees’ responses, and hardly any contradictions, which gives us, confidence in the validity of the feature-model data. Regarding the approaches for feature-model construction, the interviewees indicate that both bottom-up and top-down methods are used, depending on the interest of the client company. The two variability sources (internal variability model and product-line artifacts) are used to identify and extract features in the bottom-up approach. Both interviewees report that (a) the data type used for features is mostly Boolean, (b) alternative feature groups are used frequently, (c) default values can be specified for features, (d) soft constraints (e.g., *recommends*) can be specified, and (e) features can be annotated with attributes.

Both interviewees confirm the use of family models to represent artifacts in the solution space, and the use of simple Boolean expressions to map feature models to family models. In addition, interviewee 2 describes two distinct family-model representations for product-line artifacts. Both interviewees report

on different mechanisms used to realize variability in product-line artifacts, and confirm that #IFDEFs are the least preferred among their clients. Data from both interviews suggest that both static and dynamic configurations are used, while static configurations are the most common.

<i>Feature Model Comparison Entity</i>	<i>Interviewee 1</i>	<i>Interviewee 2</i>
<i>Size of feature model</i>	Bottom-up: on the order of a couple of 100s (500-1000) Top-down: around 100	On the order of a couple of 100s; makes no distinction between the sizes of feature models that are constructed top-down versus bottom-up; however, indicates that bottom-up is used commonly
<i>Height of the feature model</i>	3-5 levels, except one case with >10 levels	3-4 levels
<i>Type of features</i>	Mostly functional features, some quality features that impact functional features	Functional features
<i>Feature granularity</i>	Bottom-up: fine-grained technical features Top-down: more abstract features	Fine-grained technical features; makes no distinction between top-down and bottom-up feature models in terms of feature granularity
<i>Dependencies within feature model</i>	Alternative groups are frequent; OR groups less frequently used	Alternative groups are frequent
<i>Cross-tree constraints</i>	Very few cross-tree constraints; typically <i>requires</i> , <i>conflicts</i> , <i>recommends</i> relations; simple Boolean expressions or sets of Boolean expressions joined by <i>AND/OR</i>	Very few cross-tree constraints; typically <i>requires</i> , <i>excludes</i> , <i>recommends</i> relations; simple Boolean expressions or set of Boolean expressions joined by <i>AND/OR</i>
<i>Modularization of feature models</i>	One feature model in most cases, except 1 case with 5000 features, which is decomposed into 1 super model and 50 reusable sub-models	No indication if modularization of feature models is used by the client companies; but the interviewee mentions pure::variants capability for relating multiple feature models hierarchically, using the concept of configuration spaces
<i>Data type, Attribute type</i>	Mostly Boolean features; features can have attributes of enumerated type	Mostly Boolean features; features can have attributes of basic types

Table 5: Comparison of interview 1 and interview 2

4.3 Discussion of Interview Results and Findings

We have reported the processes, methods, tools, and the related practices regarding variability modelling and product-line development within six organizations. All interviewees have extensive experience (>3 years) with product lines and variability models (especially, feature models), and all of them have acknowledged the value of using feature models to represent variability. In this section, we summarize and compare the practices and identify common trends, benefits, and challenges, as well as present interesting observations. We discuss these practices with respect to the two main variability topics mentioned earlier in section 4.1: (a) variability modelling and realization, and (b) benefits, challenges and recommendations. In the following, we refer to each subject as company ‘n’ according to its respective case study number ‘n’ (for example, company 1 refers to the subject in case study 1), except for the tool builders (case study 5 and case study 6), to which we refer by their names.

Both tool builders and companies 2 and 3 assist their clients in building the clients' product lines, including the construction of variability models. Hence, their practices concern more than one product line belonging to the same or multiple domains. In contrast, companies 1 and 4 have developed their own individual product lines, transitioning from a traditional software-development approach to a product-line approach and transferring most of their artifacts to one or more product-line platforms. Since these two companies are the only subjects who developed their own product lines, we have detailed information on only their product-line adoption strategies and product structure; this information is rather limited or is unavailable for other companies who assist their clients, especially with respect to feature-model construction and variability realization.

A. Variability modelling and Realization

Below, we compare the companies' approaches to product-line adoption, variability-model creation, modularization and structuring of variability models, evolution of variability models, variability realization in artifacts, and the mapping of artifacts to variability models.

Practices on product-line adoption:

Developers at company 1 familiarized themselves with the research on product-line development and variability modelling, mainly through attending workshops and conferences. This company followed a systematic approach to product-line development, and has achieved a 100% platform for all of its products. They have written about their success stories and have disclosed those practices that aided them to achieve better results. A few of their success factors include: (i) a separate product-line team dedicated to ensuring that all products are built from reusable artifacts and are supported by the product-line platform; and (ii) a strong symbiotic relationship between the product-line team and the development teams.

Similar to company 1, company 2's developers studied existing product-line development methods and processes, and followed a systematic approach to product-line development. Unlike company 1, which has many divisions and levels of authority, company 2 is a small independent company that owns a small product line (40 features), which is used internally within the organization to assist their clients. The development is done mostly by a team of two people and therefore, in our opinion, their transformation to a product-line approach was relatively easy. For instance, the interviewee suggests that they did not even consider splitting into multiple teams; whereas they recommend division of responsibilities in larger companies. In contrast to companies 1 and 2, company 4's decision to adopt product lining was not a conscious effort, but rather an evolutionary process that happened through the course of 10-15 years. As mentioned earlier, interviewees from both tool builders and company 3 do not have detailed insight into the process that motivated these companies' clients to adopt a product-line approach.

Practices on variability-model construction process:

All of the interviewees' companies use feature modelling in their product-line projects, except company 3, which uses feature models in only one of its projects. This confirms that feature modelling is the most popular notation used for variability modelling; hence, our discussion on variability modelling is confined to feature models. Please see Table 6 for a summary of variability models, tools, types of product-line artifacts, and variability realization mechanisms used by the six companies.

The information available on the feature-model creation process is more elaborate for the tool-builder companies, compared to the available information about the other subjects' processes. The reasons for this may be that the main focus of the tool builders is to assist their clients with feature-model construction by using their respective tool frameworks. Developers from both tool builders follow two similar generic approaches to feature identification and feature-model construction for their clients: (a) a top-down approach that constructs a feature model from (mostly) requirements, and (b) a bottom-up approach that builds a feature model from existing solution-based artifacts. We did not find any significant differences between the two tool builders in the usage of these approaches, despite the mention of an additional source to build feature models for the bottom-up approach by pure::variants (interview 2). It is interesting to note that one of the interviewees from pure::variants reports that the success of the two techniques is based on who within their client companies handles the process of feature-model construction.

Developers from company 2 build their own internal feature models to represent the variabilities that exist in their clients' applications. The features are identified based on discussions with the clients. Although this company does not share its feature models with its clients, it uses the feature models to help clients in managing and deriving their products. In the case of company 1, there is a domain expert responsible for building the feature model. Similar to company 2, company 1's feature model is built through discussions with different development teams, and they use the feature model mainly for the automatic derivation of products. Although, company 1's interviewee did not provide information on the feature-model creation process, we were able to learn from one of the company's publications [57] that they typically use the bottom-up approach.

Company 3 uses feature models as a means to express component variabilities better, in order to overcome modelling difficulties when using textual DSLs. Hence, the features are fine-grained and are matched mostly to component functionalities. Recall that the company developed a feature model for the insurance project and used DSLs to model variability in all other projects. The above observation suggests that company 3, having more expertise in DSL, adopted feature modelling to better represent certain kind of variabilities that were otherwise less expressive or more difficult to model using DSLs.

Although the interviewee from company 4 says that they use feature models, we are not sure if the interviewee refers to a graphical feature model versus a database containing a list of *optional* and *mandatory* features. Our suspicions are based on certain company practices, as reported by the interviewee, which include: (1) absence of feature grouping, (2) absence of a feature-modelling tool, (3) absence of any formal structural description of the feature models, as well as their mappings to artifacts, and (4) absence of any formal documentation available for feature models. It is possible that the top-level feature database is a list of the high-level (abstract) features, maintained by the centralized team; that the intermediate-level feature database is a list of relatively less-abstract features used for sub-system classification, maintained by an intermediate development team; and that the low-level feature databases are lists of leaf-level sub-system features, maintained by the respective sub-system teams.

Practices on modularization and structuring of feature models:

All companies used a single feature model to express their product-line variabilities. Multiple feature models are only used by company 4 (at least three different levels: top, intermediate, and low), company 5 (especially for those clients with multiple sub-systems), and in an exceptional case reported by the interviewee from pure::variants. When multiple feature models are employed, the number of features in the product line is much larger than the numbers of features in product lines that have a single feature model. For instance, the largest single feature model has a total of 1100 features, whereas in the modularized feature models, there is on the order of hundreds of features in each feature module, and several thousands of features in the full product line. It is interesting to note that company 1 has expressed interest in modularizing its feature model to support different product types; the hold back is in developing a good modularized structure.

The average height of the companies' feature models is between 3-5 levels, and most of the companies prefer to limit the height of their feature models to this average limit. For example, company 1 prefers to avoid greater heights in order to keep feature models simple. Interestingly, one of the pure::variants interviewees advises against flat feature models, which is not the case for most of his clients. One possible reason for advising against flat feature models could be that they lead to additional cross-tree constraints, which complicates the models.

All companies model mostly functional features. There is no report of nonfunctional features being captured in any feature models. One of the interviewees from pure-systems makes an interesting remark in this regard, which suggests that there is no distinction between nonfunctional and functional features, and that nonfunctional features with a direct impact on functional features should be modelled.

Company 1 uses strategies to organize the features within a feature model, such as placing product-specific features in one tree branch and common features in another branch. This practice might help

developers to more easily identify parts of the feature model that might be affected when a new product is added or when existing features are refactored. Company 2 also follows a similar systematic research-based approach to arranging features in their feature model. No information is available regarding the feature-organization methods used by companies 3 and 4. The interviewees from the tool-builder companies discussed at length how they assist their clients in organizing, naming, and configuring features, by way of capabilities offered by their respective tool frameworks. In particular, both tool frameworks support multiple feature views, pre-defined feature dependency and constraint relations, and data types for features.

Trends on feature-modelling dependencies, cross-tree relations, data-types:

From Table 6, it is clear that feature grouping is used often by all companies, with *OR* and *XOR* groups being used most frequently. The only exception is company 4, which to our knowledge does not use any feature-modelling notation or tools. Boolean is the most commonly used feature type, presumably because most variabilities represent whether an *optional* feature is present in a product. Company 1 also occasionally uses string feature types. Although both tool frameworks from the tool builders support all basic feature types, there is no indication that these are actually used by their clients. Both interviewees from pure::variants say that some clients prefer to attach attributes to features rather than represent structural variabilities as sub-features (e.g., modelling CPU type as an attribute rather than as a sub-feature). In this manner, features are used only to represent clear functionality or sub-functionality, and not just any characteristic of a feature.

All of the interviewees' companies limit their use of cross-tree constraints within feature models in order to keep the models simple. The types of cross-tree relations used are typically simple Boolean expressions (e.g., *requires*, *excludes*, *conflicts*, *recommends*) or multiple Boolean expressions joined by *AND/OR* relations. Companies 2 and 4 do not use any cross-tree constraints at all; instead, company 4 expresses constraints between artifacts and feature models using informal annotations. The tool used by company 2 does not support cross-tree constraints. Company 1 and the tool builders expressed interest in documenting soft constraints that reflect how the selection of one feature is preferred to other feature selections. The use of soft constraints is a good practice, since these constraints help developers to obtain better feature configurations. pure::variants provides automated support for checking that a configuration satisfies feature constraints, but we do not know if this capability is appreciated or is even used by the company's clients. Companies 1 and 3 use pure::variants, but they did not indicate any usage of this capability --- perhaps because they are unaware of this ability, or they are not interested in generating configurations from the feature model, or users find it hard or too complex to apply these capabilities.

Practices on evolution of feature models:

The interviewees from companies 1, 2, 4 and from the tool builders commented on the evolution of feature models and artifacts. Feature addition is the most common type of evolution in these companies. Company 1 also performs feature removals, and company 4 performs refinement of existing features. Both of these companies update their feature model as the product line evolves: company 1 inspects and reviews updates to the feature model, whereas company 4 employs a change-management update process.

It is natural that feature additions are the most common type of evolution, given that most companies extend their product lines by adding new products or extensions to products. However, the interval and the number of additions vary between and within companies. Product-line evolution typically does not cause major changes in the hierarchical structure of feature models. However, one of the GEARS's clients had to organize its feature sets frequently by renaming features, or using alternative ways of ordering features in the feature-model hierarchy, or modifying feature types associated with features (few details are available about the restructuring process used by this client).

Practices on variability realization in artifacts and mapping of artifacts to variability models:

Code-based artifacts (usually, C, C++, or Java files) are the most common type of solution artifact used by all companies. In addition, company 2 maintains variable HTML, CSS, and PHP scripts, because this company is focused on variants of web-based applications. Among all subjects, company 4, being a large-scale automotive company that builds many sub-systems and frequently changing features, uses a large number of variable artifacts, including logic design blocks, components models, Simulink models, AUTOSAR models, and code-based artifacts.

#IFDEF directives and IF-statements are the most commonly used variability realization mechanism in code artifacts. Most of the companies prefer to limit their use of #IFDEFs for various reasons. For instance, developers from company 1 reported that too many #IFDEFs affect code readability, and many of pure::variants' clients mentioned having bad experiences with #IFDEFs (details are not available). It is interesting to note that pure::variants' clients opted for dynamic binding of configuration files (using optimizable configuration parameters) rather than static #IFDEFs. The second most common variability-realization mechanism used in implementations is configuration files, which are maintained in a central database and contain many adaptable parameters (e.g., 270 configuration parameters in the case of company 4).

Companies 1 and 2, and most of tool builders' clients formally mapped features to artifacts. We believe that a formal mapping is essential only if a company plans to automatically generate feature configurations and derive products, as in the case of company 1. In the case of company 3, their feature

model is used to express and understand their existing DSL-based variabilities, and therefore family models are not used even though they are supported in the feature-modelling tool. Similarly, in the case of company 4, feature models are used to understand product scope and functionalities and to associate features to artifacts; hence, the company uses informal annotations that link features to certain design artifacts. However, the interviewee reports that the company is interested in formalizing the annotations, so that it can progress towards automatic generation of products from feature models.

All six companies support mostly static configurations of their product-line projects. Companies 4, 5, and 6 sometimes use dynamic binding. Note that company 4 and some of the tool builders' clients work in the automotive domain, where dynamic or run-time variability is essential, especially during start-up.

Variability Modelling and Realization	<i>Case study 1</i>	<i>Case study 2</i>	<i>Case study 3</i>	<i>Case study 4</i>	<i>Case study 5 (GEARS)</i>	<i>Case study 6 (pure systems)</i>
<i>Variability modelling notation</i>	feature modelling	feature modelling and DSL	feature modelling and DSL	feature modelling, variability constructs in UML, Vector-EC, Simulink	feature modelling	feature modelling
<i>Units of variability</i>	features	features	features	features	features	features
<i>Number of features/variability units</i>	1,100 features (product line for frequency converters)	40 features (product line for web applications)	100 features (product-line for insurance system)	300–500 features for the top-level feature model, 300–800 for each subsystem fragment (product line for car models)	on the order of hundreds	on the order of hundreds (Interviewee 1 mentions around 100 features for top-down approach)
<i>Model height (number of levels)</i>	3–4	5–6	3–4	2–3	< 3	3–5, except for 1 case with >10 levels
<i>Dependencies within feature model</i>	alternative groups: <i>OR</i> , <i>XOR</i> , <i>mandatory</i> , <i>optional</i>	alternative groups: <i>OR</i> (frequently used), <i>XOR</i> , <i>mandatory</i>	alternative groups: <i>OR</i> (frequently used), <i>XOR</i> (frequently used), <i>mandatory</i> , <i>optional</i>	<i>mandatory</i> , <i>optional</i> , No feature grouping	mentions that <i>mandatory</i> , <i>optional</i> , and groups are supported by the tool; does not mention which of these are used most often by clients	alternative groups (<i>OR</i> groups are used less often)
<i>Cross-tree constraints</i>	very few relations, uses <i>recommended</i> relation	not represented in feature model	<i>requires</i> , <i>excludes</i>	not represented in feature model	very few cross-tree constraints; in the form of assertions expressed using Boolean expressions	very few cross-tree constraints; typically <i>requires</i> , <i>conflicts</i> , <i>excludes</i> , <i>recommendations</i> relations; single or sets of Boolean expressions joined by AND/OR

Table 6: Details on variability models: tools used, types of product-line artifacts, and variability-realization mechanisms

Variability Modelling and Realization	<i>Case study 1</i>	<i>Case study 2</i>	<i>Case study 3</i>	<i>Case study 4</i>	<i>Case study 5 (GEARS)</i>	<i>Case study 6 (pure systems)</i>
<i>Modularization of feature models</i>	1 feature model	not mentioned if multiple models are present; mentions details mainly regarding single feature model for clients	1 feature model for clients until now	1 top-level feature model, 1 intermediate feature model; several lower-level sub-system feature models	multiple sub-system feature models for certain clients; uses feature profiles to configure multiple models	mostly one feature model, except in one case with 5000 features in which the feature model is decomposed into 1 super model and 50 sub-models; No indication that modularization of feature models is used by the client companies
<i>Variability modelling and realization tools</i>	pure::variants	CaptainFeature, EMF, an Eclipse plugin, XFramer	pure::variants	TeamCenter, Vector-EC, UML, Simulink, KDP configurator	GEARS	pure::variants
<i>Feature data type, Attribute type</i>	95% Boolean features, a few string features; default values for features supported	mainly Boolean features; default values not supported	not explicitly mentioned; talks about features as a nice way of organizing Boolean configuration switches	mainly Boolean features; integer features not supported	mentions that all basic data types for features are supported by the tool, including integer, Boolean, float, and string; does not mention which of these are used often by clients	mainly Boolean features; attributes are attached to features

Table 6 (continued): Details on variability models: tools used, types of product-line artifacts, and variability-realization mechanisms

Variability Modelling and Realization	<i>Case study 1</i>	<i>Case study 2</i>	<i>Case study 3</i>	<i>Case study 4</i>	<i>Case study 5 (GEARS)</i>	<i>Case study 6 (pure systems)</i>
<i>Model evolution</i>	feature addition and removal	feature addition	details not available	feature addition and refinement	feature additions and refactoring	details not available
<i>Mapping to artifacts</i>	pure::variants' family model with complex conditions, feature Makefiles	Java-based imperative scripts (within generator)	no explicit mapping: informal links to relate feature models to DSL models	informal text, or picture-based annotations maintained in a proprietary database, in a separate Word document, or attached to design artifacts	variation point logic; Boolean conditions	pure::variants' family model with simple Boolean conditions
<i>Binding time</i>	static	static	static	static, dynamic	static, dynamic	mostly static, dynamic too

Table 6 (continued): Details on variability models: tools used, types of product-line artifacts, and variability-realization mechanisms

B. Discussion of Benefits, Challenges and Recommendations

All of the interviewees appreciate the better visualization and product-scoping capabilities that are realized as a result of adopting feature-based variability modelling as part of their product-line development. Most of the companies are able to better understand the product functionalities in ways that was not possible before. For example, after adopting feature modelling, the developers from company 1 were able to identify duplicate implementations of certain functions and to easily understand code in terms of features. The developers from company 3 were able to identify which component(s) realized which feature(s). In addition, the interviewee from company 3 acknowledges the value of feature models in expressing certain kinds of variabilities. Product lining helped company 1 to considerably improve their quality of software, by reducing the number of critical bugs in their software. The interviewee from company 2 reported that feature-oriented product-line development helped his company's developers to obtain a better understanding of the variable options.

Despite the advantages achieved by product lining, company 1 currently faces four major challenges. The primary issue is the high level of skill and mindset changes required by the developers and architects to align their existing software-design and architecture-methods to support a 100% product-line platform. A positive direction in this regard would be to train developers and architects to think at an abstract level (similar to the product-line mindset described by Chen et al. [10]). This in turn, can help them to adapt

and accommodate variations in their development methods. Secondly, the product-line team has to expend a significant amount of effort checking the validity of newly added functionalities and ideas emerging from different development teams. The third challenge is that different levels of authority have difficulty reaching consensus regarding product-line decisions. This is a crucial concern that needs to be addressed, because it affects the longevity of their product line. The management issue could be improved by exploring (1) what major communication barriers exist between different divisions, (2) why the middle or product management has difficulty in accepting variability-related decisions, and (3) what methods or factors will help the product-line team to gain the necessary support from the managers. The fourth challenge is that product-line adoption prompted the developers to test components individually using unit testing, rather than testing products as a whole; this means that components must be encapsulated and tested in isolation for run-time environments.

Company 4, being a large-scale company, also shares the above-mentioned challenges. In addition, the company's management requires the developers to use a number of different tools. We were told that the developers look forward to an integrated end-to-end tool chain that manages variabilities and assists with automatic product derivation. We consider it important to address the scalability issues that company 4 faces due to large numbers of variabilities and configurations in its product line, and the involvement of many development teams. From the perspective of this company's interviewee, the organization does not see the value of using feature modelling for variability-configuration and realization activities; this observation is interesting because the interviewee said earlier that the developers in his company are working towards mapping the feature models to different product-line artifacts (e.g., components, Simulink models). Currently, their major concern is how to manage a large number of dependencies that exist between subsystems, and how to map the feature model to different artifacts (e.g., Simulink models, code artifacts), to help with product derivation. In this regard, our opinion is that feature modelling can assist the developers to some extent by expressing the intricate dependencies or constraints between features. Products can be automatically derived from these feature configurations, if appropriate mappings are established between the feature model and artifacts using a single flexible tool or a set of interoperable tools.

The challenges experienced by companies 1 and 4 are less applicable to companies 2 and 3 because the latter are medium-scale consulting companies that own small product lines with relatively fewer features. The biggest challenge that company 2 currently faces is to have an adequate, cost-effective, single flexible tool or tool chain for variability that can provide better visualization, user interfacing, configuration, and evolution capabilities.

4.4 Threats to Validity

Our results are solely dependent on the perspectives of the interviewees and the interview transcriptions, and are therefore subject to the usual validity threats in an interview-based qualitative study. The technical information provided by the interviewees is based on their memory, understanding, and experiences concerning the extent to which a particular practice is realized in their organizations (in particular, the estimations about the variability models). The interview from company 2 and one of the interviews (interview 1) from pure::variants were translated from German, and hence we had to rely on the translations of the quotations from these interviewees. Another main threat for this study is the limitation in the number of interviewees: one interviewee per company, except for pure::variants with 2 interviewees. To compensate for this limitation, we carefully avoided any generalization of the interview results, and we did not consider the data provided by the interviewees as fully representative of their respective companies. We can only assure that the findings and trends hold with respect to the interviewees. In order to limit the scope of our study, we avoided some factors that could influence variability-modelling factors, such as economic or political aspects. As with any interview-based study, it is possible that we might have misinterpreted some interviewee responses, which could have affected our findings, compromising the reliability of our study.

Chapter 5

Conclusions

In this thesis, we have described the variability-modelling practices in industrial product lines using a qualitative study involving two empirical methods: a questionnaire-based survey and semi-structured interviews. We have investigated companies' variability-modelling practices and experiences with the aim to gather information on (1) the methods and strategies used to create and manage variability models, (2) the tools and notations used for variability modelling, (3) the perceived value and challenges of variability modelling, and (4) the core characteristics of companies' variability models. While the survey (35 responses) provided us with a preliminary record of industrial practices in variability modelling, the interviews (7 interviews) helped us to obtain a clear and detailed qualitative record of actual practices and of proprietary variability models.

Our survey results show that variability models are often created by re-engineering existing products into a product line. We found that only a small number of our study participants chose the systematic proactive approach to product-line development; this finding suggests the need to channel product-line research towards processes, methods, strategies, and tools that are suitable for systematically re-engineering existing systems into a product-line.

Both survey and interview results indicate that developers use variability models for many purposes, such as the visualization of variabilities, configuration of products, and scoping of products. To construct variability models, the companies in our study use the top-down approach or the bottom-up approach reported in the SPL literature. However, without further investigation, we cannot establish which of these two methods is used most often or is most successful. All of the interviewees and the majority of survey participants indicated that they represent variability using separate variability models rather than annotative approaches. This finding signifies the relevance of the emerging OMG variability-modelling standard, the Common Variability Language (CVL) [48], which uses separate variability models. Although we observed both from survey and interviews that a high degree of heterogeneity exists in the notations and tools used by organizations for modelling and realizing variability, feature models and feature-based tools are the most common. We also saw huge differences in the sizes of variability models and their contents, which indicate that variability models can have different use cases depending on the organization; this suggests that variability-modelling tools need to satisfy varying requirements based on organizations' interests. We found that the majority of our study participants prefer to limit the number of cross-tree dependencies in their variability models; reports from interviews suggest that this is a good practice because it helps the developers to maintain simple feature models that are easier to view and reason.

Both survey and interview results indicate that our study participants perceive the value of variability modelling and acknowledge the use of feature models. The majority of interviewees said that feature modelling has helped developers to better understand the product functionalities and variable options. However, reports from interviews suggest that variability modelling requires developers to have expertise in specific areas, such as the identification of features, construction of variability models, and configuration of variability models. In addition, a few of the interviewees mentioned that variability modelling and product-line adoption have forced developers to think in terms of a product-line scenario rather than a product-based scenario.

The majority of our study participants reported complexity challenges that were related mainly to the visualization and evolution of variability models, and dependency management. In addition, two of the interviewees reported an organizational challenge that described the difficulty faced by the different levels of authority within their organizations to reach consensus in enforcing product-line decisions.

Although our study is able to provide an insight into the variability-modelling tools, methods, and processes used by industrial organizations, further investigation is required to obtain a deeper understanding of the variability-related practices and issues. We believe that detailed artifact studies and analyses and more semi-structured interviews involving industrial employees will provide adequate empirical data to support our findings from survey and interviews. Such detailed efforts can, (1) benefit the SPL research community by identifying research topics that might eventually help to provide successful methodologies/approaches and valuable advice to product-line organizations; and (2) help tool vendors and process designers to make informed decisions about variability modelling.

Bibliography

- [1] P. Heymans and J.C. Trigaux, “Software product lines: State of the art”, *Technical Report for PLENTY project*, EPH3310300R0462 /215315, Institut d'Informatique FUNDP, Namur, 2003. Available at: <http://www.inf.ufpr.br/silvia/topicos/artigostrab10/artigo1-S1e2.pdf>
- [2] Thomas Stahl, Markus Voelter and Krzysztof Czarnecki, “Model-driven software development: Technology, Engineering, and Management”, *John Wiley & Sons*, 2006.
- [3] A. Birk, “Product line engineering, the state of the practice”, *IEEE Software*, vol. 20, no.6, 52-60, 2010.
- [4] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “The variability model of the Linux kernel”, *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010.
- [5] R. Flores, C. Krueger, and P. Clements, “Mega-scale product line engineering at general motors”, *Proceedings of the Software Product Line Conference (SPLC)*, 2012.
- [6] H. P. Jepsen and D. Beuche, “Running a software product line: standing still is going backwards”, *Proceedings of the Software Product Line Conference (SPLC)*, 2009.
- [7] Marco Sinnema, Sybren Deelstra, Jos Nijhuis and Jan Bosch, “COVAMOF: A framework for modelling variability in software product families”, *Proceedings of the Software Product Line Conference (SPLC)*, 2004.
- [8] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans, “A preliminary review on the application of feature diagrams in practice,” *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2010.
- [9] L. Chen and M. A. Babar, “A systematic review of evaluation of variability management approaches in software product lines”, *Information and Software Technology*, vol. 53, no. 4, 344 – 362, 2011.
- [10] L. Chen, M. A. Babar, and N. Ali, “Variability management in software product lines: a systematic review”, *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009.

- [11] L. Chen and M. Ali Babar, “A survey of scalability aspects of variability modelling approaches”, *Workshop on Scalable Modelling Techniques for Software Product Lines at SPLC*, 2009.
- [12] IDI Integrity, “Terminology”, 2006.
1 Available at: <http://www.idi-software.com/resources/defns.html>
- [13] P. Clements and L. Northrop, “A Framework for Software Product Line Practice”, *Technical Report, Software Engineering Institute: Carnegie Mellon University, Pittsburgh, PA*, Online Version 4.1, 2003.
Available at: <http://www.sei.cmu.edu/plp/framework.html>
- [14] Klaus Pohl, Gunter Bockle and Frank van der Linden, “Software product line engineering: Foundations, principles, and techniques”, *New York: Springer-Verlag*, 2005.
- [15] David M. Weiss and Chi Tau Robert Lai, “Software product-line engineering: A family-based software development process”, *Addison-Wesley Professional*, 1999.
- [16] M. Rick Rabise, “A User-Centered Approach to Product Configuration in Software Product Line Engineering”, *PhD dissertation, Christian Doppler Laboratory for Automated Software Engineering*, 2009.
Available at: http://ase.jku.at/phdtheses/Dissertation_Rabiser_Abstract.pdf
- [17] E. Dolstra, G. Florijn and E. Visser, “Timeline variability: The variability of binding time of variation points”, *Proceedings of the Workshop on Software Variability Management (SVM)*, 2003.
- [18] N. Holtz and W. Rasdorf, “An evaluation of programming languages and language features for engineering software development”, *Engineering with Computers Journal*, vol. 3, 183–199, 1988.
- [19] Manfred Broy, Ingolf H. Krüger and Michael Meisinger, “Quality assurance and certification of software modules in safety critical automotive electronic control units using a CASE-tool integration platform”, *Lecture Notes in Computer Science*, vol. 4147, 15-30, 2006.

- [20] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki and Andrzej Wasowski, “A survey of variability modelling in industrial practice”, *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2013.
- [21] C. W. Krueger, “Easing the transition to software mass customization”, *Proceedings of the Software Product Family Engineering Conference (PFE)*, 2001.
- [22] R. Johnson, *J2EE Development without EJB*. Wiley, New York, 2004.
- [23] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” *Software Engineering Institute: Carnegie Mellon University, Pittsburgh, PA, Technical Report, CMU/SEI-90-TR-21*, Nov. 1990.
- [24] K. Czarnecki and U. W. Eisenecker, “Generative programming: methods, tools, and applications”, *Addison-Wesley Professional*, Boston, MA, 2000.
- [25] L. Chen, M. Ali Babar, and C. Cawley, “A status report on the evaluation of variability management approaches,” *Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering (EASE)*, 2009.
- [26] D. Dhungana, P. Gräijnbacher, and R. Rabiser, “The dopler meta-tool for decision-oriented variability modelling: a multiple case study,” *Automated Software Engineering*, vol. 18, no. 1, 77 – 114, 2011.
- [27] K. Schmid, R. Rabiser, and P. Gräijnbacher, “A comparison of decision modelling approaches in product lines,” *Proceedings of the 5th Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2011.
Available at: <http://dl.acm.org/citation.cfm?id=1944907>
- [28] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite, “From goals to high-variability software design,” *Proceedings of the 17th International Conference on Foundations of Intelligent Systems (ISMIS)*, 2008.
Available at: <http://dl.acm.org/citation.cfm?id=1786476>
- [29] F. Semmak, “Supporting variability in goal-based requirements,” *Proceedings of the 3rd International Conference on Research Challenges in Information Science (RCIS)*, 2009.

- [30] M. Clauss, "Generic modelling using UML extensions for variability," *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages*, 2001.
Available at: <http://www.citeulike.org/group/1374/article/3944308>
- [31] J. Edson Alves de Oliveira, I. M. S. Gimenes, E. Hatsue Moriya Huzita, and J. Carlos Maldonado, "A variability management process for software product lines," *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, 2005.
Available at: <http://dl.acm.org/citation.cfm?id=1105651>
- [32] M. Voelter, "Using domain specific languages for product line engineering," *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009.
Available at:
<http://dl.acm.org/citation.cfm?id=1753294&dl=ACM&coll=DL&CFID=133707256&CFTOKEN=70782386>
- [33] M. Becker, "Towards a general model of variability in product families," *Proceedings of the 1st Workshop on Software Variability Management (SVM)*, 2003.
- [34] K. Pohl and T. Weyer, "Software product line engineering: Foundations, Principles and Techniques", *Springer*, 2005.
- [35] Pourya Shaker, "Feature-oriented requirements modelling", *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, ACM Digital Library, vol.2, 365-368, 2010.
- [36] M. Griss, J. Favaro and M. D'Alessandro, "Integrating feature modelling with the RSEB", *Proceedings of the Fifth International Conference on Software Reuse*, 1998.
- [37] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, "FORM: A Feature-oriented reuse method with domain-specific reference architectures", *Annals of Software Engineering*, vol. 5, 143-168, 1998.
- [38] K.C. Kang, K. Lee and J. Lee, "FOPLE – Feature-oriented product line software engineering: Principles and Guidelines", *Pohang University of Science and Technology*, 2002.

- [39] T. J. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick and C. Gillan, "Weaving behaviour into feature models for embedded system families", *Proceedings of 10th International Software Product Line Conference (SPLC)*, 2006.
- [40] D. Fey, R. Fajta and A. Boros, "Feature modelling: A meta-model to enhance usability and usefulness ", *Proceedings of the International Software Product Line Conference (SPLC)*, *Lecture Notes in Computer Science*, vol. 2379, 198-216, 2002.
- [41] Matthias Riebisch, Kai Böllert, Detlef Streitferdt and Ilka Philippow, "Extending feature diagrams with UML multiplicities", *Proceedings of the 6th International Conference on Integrated Design and Process Technology (IDPT)*, 2002.
- [42] Matthias Riebisch, Detlef Streitferdt and Ilian Pashov, "Modelling variability for object-oriented product lines", *Communication Abstractions for Distributed Systems Workshop (ECCOP)*, *Lecture Notes in Computer Science*, vol. 3013, 165-178, 2004.
- [43] H. Ye and H. Liu, "Approach to modelling feature variability and dependencies in software product lines", *IEEE Software Journal*, vol. 152, 101-109, 2005.
- [44] Krzysztof Czarnecki and Chang Hwan Peter Kim, "Cardinality-based feature modelling and constraints: A progress report", *Proceedings of Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.
- [45] K. Czarnecki, "Mapping features to models: A template approach based on superimposed variants", *Proceedings of 4th International Conference of Generative Programming and Component Engineering*, 2005.
- [46] M.O. Reiser and M. Weber, "Multi-level feature trees: A pragmatic approach to managing highly complex product families", *Requirements Engineering Journal*, vol. 12, 57-75, 2007.
- [47] J. Lee, K.C. Kang, "A Feature-oriented approach to developing dynamically reconfigurable products in product line engineering", *Proceedings of 10th International Software Product Line Conference (SPLC)*, 2006.
- [48] OMG, "Common Variability Language (CVL)" *OMG Initial Submission*, Proposal no: 091209, 2010.
Available at: <http://www.omgwiki.org/variability/doku.php>

- [49] O. Haugen, "CVL - Common Variability Language - a generic approach to variability for OMG standardization", 2010.
- [50] Charles W. Krueger, "The BigLever software GEARS systems and software product line lifecycle framework", *Proceedings of International Software Product Line Conference (SPLC)*, 2010.
- [51] D. Beuche, "Variant management with pure::variants", *Technical White Paper, Pure-Systems GmbH*, Magdeburg, Germany, 2003.
Available at: <http://www.puresystems.com>
- [52] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development", *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.
- [53] D. Batory, "Feature-oriented programming and the AHEAD tool suite", *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, 2004.
- [54] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: feature modelling plug-in for eclipse", *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology (eXchange)*, 2004.
- [55] D. Beuche, H. Papajewski and W. Schröder-Preikschat, "Variability management with feature models", *Science of Computer Programming Journal*, vol. 53, no. 3, 333-352, 2004.
- [56] Marcilio Mendonca, Moises Branco and Donald Cowan, "S.P.L.O.T. - Software Product Lines Online Tools", *In Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.
- [57] H. Jepsen and D. Beuche "Minimally Invasive Migration to Software Product Line," *Proceedings of 2007 International Software Product Line Conference (SPLC)*, 2007.
- [58] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki and Andrzej Wasowski, "Variability Modelling in the wild", *Presentation at the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2013.
Available at: http://gsd.uwaterloo.ca/sites/default/files/2013-vamos-survey_slides.pdf

- [59] Charles Krueger, “Eliminating the adoption barrier”, *IEEE Software Journal*, vol. 19, no. 4, 29-31, 2002.
- [60] Don Batory and Sean O’Malley, “The design and implementation of hierarchical software systems with reusable components”, *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, 355-398, 1992.
- [61] Eric Dashofy, David Garlan, André van der Hoek and Bradley Schmerl, “xArch”, Contributors: University of California, Irvine and Carnegie Mellon University, 2001.
Available at: <http://www.isr.uci.edu/architecture/xarch/>
- [62] D. Garlan, R. Monroe and D. Wile, “Acme: an architectural description interchange language”, *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, 169-183, 1997.
- [63] R.Allen and D.Garlan, “A formal basis for architectural connection”, *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, 213-249, 1997.
- [64] R.van Ommering, F. van der Linden, J. Kramer and J. Magee, “The Koala component model for consumer electronics software”, *IEEE Computer Journal*, vol. 33, no. 3, 78-85, 2000.
- [65] T. Asikainen, T. Soininen and T. Männistö, “Towards managing variability using software product family architecture models and product configurators”, *Proceedings of Software Variability Management Workshop*, 2004.
- [66] R. Johnson, “J2EE development without EJB”, *Wiley publications*, New York, 2004.
- [67] A. Aldazabal and S. Erofeev, “Product line unified modeler (plum)”. 2007.
Available at:
http://www.tecnalia.com/images/stories/Areas_de_negocio/TICSs_ESI/Software/PLUM/PLUM.pdf
- [68] S. Mann and G. Rock, “Dealing with variability in architecture descriptions to support automotive product lines”, *Proceedings of the 2009 International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2009.

- [69] A. Abele, R. Johansson, H. Lonn, Y. Papadopoulos, M. Reiser, D. Servat, M. Tornngren, and M. Weber. “The CVM framework: A prototype tool for compositional variability management”, *Proceedings of the 2010 International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010.
- [70] R. Bonifacio, L. Teixeira, and P. Borba and “Hephaestus: A tool for managing SPL variabilities”, *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS) Tools Session*, 2009.
- [71] R. Stoiber and M. Glinz, “Modelling and managing tacit product line requirements knowledge”, *Proceedings of the 2009 Second International Workshop on Managing Requirements Knowledge (MARK’09)*, 2009.
- [72] M. Mendonca, A. Wasowski, and K. Czarnecki, “SAT-based analysis of feature models is easy”, *Proceedings of 2009 International Software Product Line Conference (SPLC’09)*, 2009.
- [73] M. Svahnberg, J.V. Gurf, Jand J. Bosch, “A taxonomy of variability realization techniques”, *Research Articles-Software Practices Experiences*, vol. 35, 705-754, 2005.
- [74] Hongyu Zhanga and Stan Jarzabek, “XVCL: a mechanism for handling variants in software product lines”, *Proceedings of Software Variability Management Conference (SVM)*, vol. 53, no. 3, 381-407, 2004.
- [75] K. Pohl, C. A. Rummler, V. Gasiunas, N. Loughra, H. Arboleda, F. D. AFernandes, J. Noyé, A. Núñez, R. Passama, J.C Royer and M. Südholt, “Survey of existing implementation techniques with respect to their support for the practices currently in use at industrial partners”, *AMPLE Project Deliverable D3.1*, 2007.
- [76] Ivar Jacobson, Martin Griss and Patrik Jonsson, “Software reuse: architecture, process and organization for business success”, *Addison-Wesley Professional: First Edition*, 1997.
- [77] C. Gacek and M. Anastasopoulos, “Implementing product line variabilities”, *SIGSOFT Software Engineering Notes*, vol. 26, 109-117, 2001.
- [78] S. D. Kettemann, D. Muthig and M. Anastasopoulos, “Product line implementation technologies: component technology view”, *Fraunhofer IESE*, Report no: 015.03/E, 2003.

- [79] Captain Feature
Available at: <http://sourceforge.net/projects/captainfeature/>
- [80] T. Stahl, M. Volter, S. Efftinge, and A. Haase, "Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management", *Dpunkt Verlag, Heidelberg second edition*, 2007.
- [81] UNL/FCT, "Traceability Requirements", *Aspect-Oriented Model-Driven Product Line Engineering (AMPLE)*, *AMPLE Internal Documentation*, 2007.
- [82] T. Syrjanen, "Including diagnostic information in configuration models", *Proceedings of the First International Conference on Computational Logic, Lecture Notes in Computer Science*, vol. 1861, 837- 851, 2000.
- [83] TeamCenter PLM software
Available at: http://www.plm.automation.siemens.com/en_us/products/teamcenter/
- [84] Alexander Felfernig, Gerhard E. Friedrich and Dietmar Jannach, "UML as domain specific language for the construction of knowledge-based configuration systems", *International Journal of Software Engineering and Knowledge Engineering*, 2000.
- [85] Klaus Schmid and Isabel John, "A customizable approach to full lifecycle variability management", *Science of Computer Programming*, vol. 53, no. 3, 259-284, 2004.
- [86] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modelling in the systems software domain," *Generative Software Development Laboratory, University of Waterloo*, Technical Report, GSDLAB-TR 2012-07-06, 2012.
- [87] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of conguration challenges in Linux and eCos", *Proceedings of the 2012 International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2012.
- [88] J. Bayer, T. Forster, T. Lehner, C. Giese, A. Schnieders, and J. Weiland, "Process family engineering in automotive control systems: a case study", *Proceedings of 1st Generative Programming and Component Engineering Workshop for QoS Provisioning in Distributed Systems (GPCE)*, 2006.

- [89] S. Thiel, S. Ferber, T. Fischer, A. Hein, M. Schlick, and R. Bosch, "A case study in applying a product line approach for car periphery supervision systems", *Proceedings of In-Vehicle Software Conference (SAE)*, 2001.
- [90] Krzysztof Czarnecki, Simon Helsen and Ulrich Eisenecker, "Staged configuration through specialization and multi-level configuration of feature models", *Proceedings of Software Process Improvement and Practice*, vol. 10, no. 2, 143-169, 2005.
- [91] K.Czarnecki, S. Helsen, U. Eisenecker, "Staged configuration using feature models", *Proceedings of 3rd International Software Product Line Conference (SPLC), Lecture Notes in Computer Science*, vol. 3154, 266–283, 2004.
- [92] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, 471-516, 2007.
- [93] M. Verlage and T. Kiesgen, "Five years of product line engineering in a small company," *Proceedings of the 27th International Software Engineering Conference (ICSE)*, 2005.
- [94] C. Thorn, "Current state and potential of variability management practices in software-intensive SMEs: Results from a regional industrial survey," *Information and Software Technology*, vol. 52, no. 4, 411-421, 2010.
- [95] C. Thorn and T. Gustafsson, "Uptake of modelling practices in SMEs: Initial results from an industrial survey," *Proceedings of the 2008 International Workshop on Models in Software Engineering*, 2008.
- [96] I. John, P. Knauber, D. Muthig, and T. Widen, "Qualifikation von kleinen und mittleren unternehmen (kmu) im bereich software variantenbildung," *Fraunhofer IESE*, Technical report IESE-026.00/D, 2001.
- [97] J. K. Bergey, G. Chastek, S. G. Cohen, and P. Donohoe, "Software product lines: Report of the 2010 U.S. army software product line workshop," *Software Engineering Institute, Carnegie Mellon*, Technical report CMU/SEI-2010-TR-014, 2010.
- [98] F. J. van der Linden, K. Schmid, and E. Rommes, "Software product lines in action: The best industrial practice in product line engineering", Springer-Verlag.

- [99] Software Engineering Institute, "Catalog of software product lines".
Available at: <http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm>
- [100] P. Grunbacher, R. Rabiser, D. Dhungana, and M. Lehofer, "Model-based customization and deployment of Eclipse-based tools: Industrial experiences", *Proceedings of 2009 Automated Software Engineering Conference (ASE)*, 2009.
- [101] M. Reiser, R. Tavakoli, and M. Weber, "Unified feature modelling as a basis for managing complex system families", *Proceedings of the 2007 International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2007.
- [102] C. Gillan, P. Kilpatrick, I. Spence, T. Brown, R. Bashroush and R. Gawley, "Challenges in the application of feature modelling in fixed line telecommunications", *Proceedings of the 2007 International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2007.
- [103] P. Pohjalainen, "Bottom-up modelling for a software product line: An experience report on agile modelling of governmental mobile networks", *Proceedings of the 2011 15th International Software Product Line Conference (SPLC)*, 2011.
- [104] C. Tischer, A. Muller, T. Mandl, and R. Krause, "Experiences from a large scale software product line merger in the automotive domain", *Proceedings of the 2011 15th International Software Product Line Conference (SPLC)*, 2011.
- [105] H. Gustavsson and U. Eklund, "Architecting automotive product lines: Industrial practice", *Proceedings of the 14th International Software Product Line Conference: going beyond, (SPLC)*, 2010.
- [106] Bernd Hardung, Thorsten Kolzow and Andreas Kruger, "Reuse of software in distributed embedded automotive systems", *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, 2004.
- [107] C. Dziobek, J. Loew, W. Przystas, and J. Weiland, "Handling functional variants in Simulink models," *Proceedings of Mathworks Automotive Conference (MACDE)*, 2008.
- [108] DSM community, "The Design Structure Matrix (DSM)".

Available at:

<http://www.dsmweb.org/en/understand-dsm/tutorials-overview/descripton-design-structre.html>

Appendix A

A Snapshot of Survey Design using the SurveyGizmo tool [58]

7-Minute-Questionnaire on Industrial Use of Variability Modeling

Distributed to over 60 practitioners and researchers with industrial experience

Dear participant,

thank you for taking some time to contribute to our study on industrial variability modeling. Answering this questionnaire will take you about seven minutes. It comprises questions about your experience in variability modeling, specifically, we ask for:

- the purpose of variability modeling;
- the notations and tools used;
- the scale of your models;
- modeling problems;
- the context of variability modeling (some characteristics of the product line).

Of course we assure anonymity and will treat your information confidentially. We kindly ask for your contact information (name and email address) at the end of the questionnaire for verification and analysis (e.g. to identify duplicates), and to notify you about the study results.


Thanks,

Ralf Rublack - University of Leipzig
Thorsten Berger - University of Leipzig
Dirya Nair - University of Waterloo
Martin Becker - Fraunhofer IESE
Andrzej Wasowski - ITU Copenhagen
Josanne Ailée - University of Waterloo
Krzysztof Czarnocki - University of Waterloo

Next

0%

Student research survey powered by:



Professional research tool with SPSS exports

Appendix B

Interview Guide

Questions about personal experience in variability modelling and product-line adoption

- What is your experience and background with variability modelling? (1.1)
- What uses of variability modelling do you find most valuable? (1.2)
- In how many projects have you used it? (1.3)
- What kind of projects were that? (1.4)
- Which application domain? (1.5)
- What is/was your role in this project? (1.6)
- How many succeeded or failed? (1.7)
- What were the main reasons for the failure? (1.8)

Questions about Modelling

● applications

- What the variability models are used for (variability modelling, domain modelling (including commonality modelling), marketing features scoping, product derivation; modelling of variability in requirements, design/architecture, deployment, testing, HW, ...) (2.1)

● unit of variability

- What kind of properties are captured as features/decisions? For example., in the systems software domain, we identified project-specific, build, lifecycle, deployment, hardware environment, software environment, third-party (imported variability), external libraries, and test cases. (3.1)

● orthogonality

- Are the VMs represented as separate models? (4.1)
- Criteria for decomposing into separate models? (4.2)

● data types

- What data types are supported by the tool used and how frequent are the types used? E.g., Bool vs. String (Unicode?)/Int/Float (other basic types? date?), feature groups, collections, classifiers (5.1)

● hierarchy

- Is it used? (6.1)
- How deep? Branching factor? (6.2)
- What's the semantics? (e.g., child-parent presence implications, or just visibility induced?) (6.3)
- Visible in model and configuration? (6.4)

- What are the criteria for creating hierarchy? Crosscutting vs. localized, cf. FM organization in the TSE paper (6.5)

- **dependencies and constraints**

- What are constraints used for? (7.1)
- What is the expressiveness of the constraint language? Data types, operators (7.2)
- How complex are the actual constraints (expression size and number of features involved => let's reuse the metrics we have already used in the past (7.3)
- How many constraints compared to the number of features? (7.4)
- Is there need for constraints over classifiers (quantification over classifier instances)? (7.5)
- Who is writing these constraints? Is it easy to get them right (are there frequent mistakes)? (7.6)
- Is it sometimes hard to satisfy the constraints during configuration? (7.7)
- Support for fixed and computed defaults? What percentage of features have explicitly defined defaults? Is there a default default? Are there any challenges in setting up and using defaults? (7.8)
- How are constraint violations detected and resolved with the addition/deletion of features or variants (7.9)

- **mapping to artifacts**

- Are other artifacts configured by the VMs? (traceability, actual derivation) (8.1)
- What are the types of these artifacts? e.g., Word documents, C files, UML models, test scripts, etc. (8.2)
- How are the mappings realized (annotative? compositional? generator scripts? etc.) (8.3)
- How are variation points realized (e.g., OO design patterns, config variables, preprocessor, etc.) (8.4)

- **binding time and mode**

- Is binding time/mode modelled in the VM? What times and modes are supported? Is minding mode parameterized? (9.1)
- How is binding time/mode technically realized? (9.2)
- How are conflicts in the binding times of variants resolved (staged configuration?) (9.3)

- **modularity**

- Are VMs divided into smaller chunks? (10.1)
- What is the mechanism used? (10.2)
- Is there a notion of an interface? (10.3)
- What are the decomposition criteria? (10.4)
- Are there any composition techniques supported? (e.g., reparenting, merging, etc.) (10.5)
- Are views supported? (10.6)
- Staged and multi-level configuration? (10.7)
- Workflow-based configuration? Role-based access to the models? (10.8)
- Cooperative configuration? Are there several roles involved? (10.9)

- **tool aspects**

- Are there any commercial VM tools used (Gears, pure::variants)? (11.1)
- If no commercial tools, what tools are used? (11.2)
- Rationale for tool choice (11.3)
- Does the tool support derivation process? Other product configurators? (11.4)
- Home-grown tools? (11.5)
- Spreadsheets, mind-maps? (11.6)
- What tools are used in the early phases of VM? (11.7)

- What other tools the VM tolling integrates with? e.g., Doors plugin, build scripts, etc. (11.8)

- **Granularity for representing variations**

- For each abstraction stage (design, architecture, implementation), what is the required level of granularity (detail) to represent variations (eg: fine-grained functions Vs coarse-grained functions)? (12.1)
- criteria for not decomposing further (12.1)

- **Variability Evolution Management**

- How companies managed to maintain updations (evolutions) in architecture to cope up with functional growth, cost cutting, h/w adaptations, and so on? (13.1)
- Are these architecture evolution decisions mainly driven by non-functional requirements? (13.2)
- What are the different evolution mechanisms and methods used (merging, cloning, linear evolution etc.) (13.3)

- **Types of variability**

- What common types of variabilities are often modelled by companies or are of interest to companies (behaviour variability, data type variability, attribute variability, non-functional variability, fault based variability etc.) (14.1)

Process questions

- **introducing PL and VM**

- Which PL approach: proactive, reactive, refactorive (15.1)
- How are VMs created? e.g., in variability modelling workshops, by reverse engineering from existing products, ... (15.2)

- **creating process**

- Who are the involved roles and what are their responsibilities (creating, updating, reading, configuring, etc.)? (16.1)
- Are there view-specific VMs (e.g., marketing, functional requirements, systems architecture, deployment, HW, etc.)? (16.2)
- How are these related? (16.3)
- How is the traceability and consistency managed among these views? (16.4)
- What the key challenges in adopting, creating, and maintaining VMs? (16.5)

- **evolution of models**

- How are the models evolved? (17.1)
- What is the relationship between CM and VM? (17.2)
- How is staging handled (products in maintenance, current products, future products)? Quality assurance vs. product evolution/development? (17.3)
- How versioning fits in? (17.4)
- Traceability of features across product evolution? (17.5)
- Do you refactor VMs? (17.6)
- How do you assure the correctness of refactoring?(part of question on evolution) (17.7)

- **finding features**

- How do you identify features? (18.1)
- How/Where to define granularity of artifacts and their assignment to features? (18.2)

- **sizes**

- How large do you models get?(size in the number of features?) (19.1)
- what is the number of divided products? (the number of products? other?) (19.2)
- How do you manage large models? (19.3)

- **process interaction**

- How VM processes interact with other development processes? e.g., requirements engineering, system design, testing, deployment, etc. (20.1)
- How formalized is the VM process? (20.2)
- What is the type of product structure development adopted by companies (integration based: central platform and systems/subsystems deployed on the platform, compositional based: distributed teams owning platforms)? (21.1)
- Is traceability of variations supported, if so, what are the methods/mechanisms adopted to trace variations within models/artifacts in an abstraction stage and between artifacts at different abstraction stages? (21.2)

Problems/Perception

- Problems/Challenges in adopting, creating, and maintaining models? (e.g., management needs to convince developers, additional work, inconsistencies, organizational structure) (22.1)
- How to manage large models? (22.2)
- Benefit of variability modelling/product-line adoption (22.3)
 - Developers/Managers happy?
 - Improvement or setback in productivity? How much?
 - Biggest value of VM (making things visible, explicit dependencies, configuration, avoid duplicates, improve quality)?
 - Time-to-market influenced?

Any final remarks? Any PL-related problems not yet solved? (22.4)