Multi-Level Interfaces Between Software Product Lines

Avoiding Direct Dependencies

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M. Sc. Reimar Schröter

geb. am 12.06.1985 in Lutherstadt Wittenberg

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr.-Ing. Ina Schaefer
Priv.-Doz. Dr. Rick Rabiser

Magdeburg, den 03.07.2018

Schröter, Reimar:
Multi-Level Interfaces Between Software Product Lines
Avoiding Direct Dependencies
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2018.

Abstract

From the beginning of the software-development age, the reuse of program functionality has been an essential part of the development process. Only with an efficient reuse strategy it is possible to speed up the development. This becomes even more important when the construction of similar programs is focused. One promising solution to develop similar programs is based on software product lines, which allow a developer to implement these products based on a common code base. Although the concept of software product lines is well established nowadays, the application of product-line mechanisms can become challenging especially in systems with a high complexity, such as analyses that ensure a correct product-line modeling. To tackle this complexity, the concept of multi product lines has been proposed, in which the whole system is described by a set of software product lines with interdependencies. Thus, the reuse of software artifacts reaches a new level. However, as the involved product lines of a multi product line are closely coupled, the knowledge of the whole product-line dependencies is generally needed to implement parts of the system or to analyze it. As a result, the development of a multi product line can also be tedious and error-prone.

To avoid direct dependencies between the product lines of a multi product line, we propose the concept of multi-level interfaces. Based on this, we aim to ease the development of multi product lines including the analysis and evolution. This could also gain particular importance when looking at the trend of a software ecosystem that represents a software product line of distributed organizations. For this purpose, we suggest to use different interfaces for each level of the development process, e.g., one interface for modeling and another interface for implementing the multi product line, whereas the lower level depends on the upper level. As a result, the whole set of interfaces will ensure a complete decoupling of the involved product lines so that we can ease the implementation and the analyses of the multi product line. To explore our concept, we specified and implemented an interface for the levels of product-line modeling as well as implementation and investigated the effect on analysis tasks in real-world systems. In this context, we found that the interface on the modeling level can ease the analysis of model compositions, since the analysis result of model compositions with and without the interface depends on each other. This is especially important in a scenario of product-line evolution. Similarly, we also discovered a benefit when using an interface on the implementation level. Here, the interface protects the developer from having to perform a manual code analysis of reusable program artifacts. In addition, as proof of

concept, we also investigated the behavioral level of product lines and investigate multiple ideas on how to realize such an interface. In sum, we conclude that the concept of multi-level interfaces can be used to avoid direct dependencies between the product lines of a multi product line and that it simplifies the analysis and evolution of the whole system.

Zusammenfassung

Seit Beginn der Softwareentwicklung ist die Wiederverwendung von Software ein essentieller Bestandteil des Entwicklungsprozesses. Nur durch eine effiziente Wiederverwendung kann eine Beschleunigung der Softwareentwicklung ermöglicht werden. Dieser Aspekt ist von umso größerer Bedeutung, wenn die Entwicklung auf die Erstellung von ähnlichen Programmen abzielt. Hierbei stellen Softwareproduktlinien eine vielversprechende Lösung bereit, die es einem Entwickler erlaubt, die verschiedenen Programme basierend auf einer gemeinsamen Quellcodebasis zu entwickeln. Obwohl Softwareproduktlinien heutzutage weit verbreitet und etabliert sind, können vor allem bei komplexen Systemen hohe Herausforderungen bei der Umsetzung entstehen, wie beispielsweise bei der Analyse der korrekten Modellierung. Um der Komplexität entgegenzuwirken, können Multiproduktlinien verwendet werden. Bei Multiproduktlinien handelt es sich um eine Menge von Produktlinien, die untereinander in Beziehung stehen. Mit dem Konzept der Multiproduktlinien konnte zugleich eine neue Art der Wiederverwendung erreicht werden, in der nicht nur einzelne Quellcodeartefakte sondern sogar vollständige Produktlinien verwendet werden. Jedoch besteht bei den voneinander abhängigen Softwareproduktlinien einer Multiproduktlinie zumeist eine enge Bindung, weshalb zur Implementierung und bei der Analyse des Gesamtsystems Detailwissen über alle Abhängigkeiten benötigt wird. Dieser Aspekt mindert den erwünschten Vorteil von einer Multiproduktlinie, da beispielsweise durch das benötigte Wissen die Implementierung erschwert wird und weiterhin fehleranfällig ist.

Um die Abhängigkeiten zwischen den Softwareproduktlinien einer Multiproduktlinie zu vermeiden, schlagen wir die Verwendung von sogenannten Multi-Level Interfaces vor. Die reduzierten Abhängigkeiten zielen darauf ab, die Entwicklung der Multiproduktlinie inklusive der Analyse und Evolution zu erleichtern. Wenn man zusätzlich noch den Trend zur Entwicklung von Software-Ökosystemen betrachtet, bei der die Entwicklung der Softwareproduktlinien über mehrere Organisationen verteilt ist, erlangt die Reduzierung der Abhängigkeit noch größere Bedeutung. Daher führen wir verschiedene Schnittstellen ein, die für die jeweiligen Entwicklungsebenen einer Multiproduktlinie bestimmt sind. Zum Beispiel handelt es sich hierbei um eine Schnittstelle für die Modellierungs- und Implementierungsebene, wobei die jeweils unteren Ebenen von den darüberliegenden Ebenen abhängig sind. Insgesamt beschreiben die Schnittstellen somit eine allumfängliche Sicht auf eine Softwareproduktlinie, sodass basierend auf dem Konzept der Multi-Level Interfaces die enge Bindung zwischen den

beteiligten Softwareproduktlinien einer Multiproduktlinie reduziert werden kann. In diesem Zusammenhang zielen wir darauf ab, die Entwicklung und die Analyse des Gesamtsystems zu erleichtern. Um Multi-Level Interfaces genauer zu erforschen, spezifizierten und implementierten wir eine Schnittstelle für die Modellierungs- sowie für die Implementierungsebene und untersuchten die Auswirkungen auf Analysen bezogen auf Realweltsysteme. In diesem Zusammenhang fanden wir heraus, dass unsere definierte Schnittstelle für die Modellierungsebene die Analyse von Modellkompositionen erleichtern kann. Dieser Aspekt ist besonders bei der Evolution von Softwareproduktlinien von Bedeutung, da Analysen seltener wiederholt werden müssen. Bezogen auf die Schnittstelle für die Ebene der Implementierung konnten wir ähnliche Auswirkungen feststellen. Hierbei kann die Implementierungsschnittstelle den Softwareproduktlinienentwickler dabei unterstützen, geeignete Quellcodeartefakte für die Wiederverwendung aufzudecken, ohne dass eine manuelle Analyse der Abhängigkeiten notwendig ist. Um die Allgemeingültigkeit von Multi-Level Interfaces zu untersuchen, betrachteten wir auch Schnittstellen die das Verhalten einer Produktlinie beschreiben. In diesem Zusammenhang stellten wir verschiedene Ansätze auf, um eine entsprechende Schnittstelle bereitzustellen. Anschließend untersuchen wir die Eignung der jeweiligen Ansätze. Insgesamt schlussfolgern wir, dass Multi-Level Interfaces helfen, direkte Abhängigkeiten zwischen Softwareproduktlinien einer Multiproduktlinie zu vermeiden und somit Vorteile bei der Entwicklung bezogen auf die Analyse und der Evolution des Gesamtsystems erzielt werden können.

Acknowledgments

At the beginning of this long road finalizing my thesis, it was unclear to me how many branches or directions I could have taken. Only now I know the complete length of the road and that several barriers and pitfalls exist on it. Fortunately, nobody could have told me about all that at the start, thus, I just started this long walk. In addition, I also know now that the finish line can only be reached with a strong will, appropriate support, and a little bit of luck. Since support is also necessary on different levels to achieve all desired goals, I want to use this place to express my thankfulness.

First of all, I would like to thank Gunter Saake for the supervision and the important advices he gave to me during the complete time as student as well as researcher. In this context, I also like to express my gratitude to the guys who guided me on this road and showed multiple directions to me for interesting research areas. Particularly, Mario Pukall who gave me the chance as student to support him in a challenging and highly interesting research project. I learned a lot in this project but I also had a funny time with Alexander Grebhahn, who was the second student in this research project. Other insights into further interesting projects I got from Martin Schäler and Sandro Schulze so that the final decision to take this road was inevitable. So, thank you guys for your insights! In addition to the starting point, the finish line has similar importance. Therefore, I would like to thank my external reviewers Prof. Dr.-Ing. Ina Schaefer and Priv.-Doz. Dr. Rick Rabiser who always highlighted their interest to my research topic so that it was possible for them to be my reviewers. But also during the complete time as researcher, I got the best support I could wish. For instance, a finalization of my thesis would not have been possible without my office colleagues, David Broneske and Wolfram Fenske. Thank you for the great time! Another great part for a successful thesis finalization was taken over by Thomas Thüm. Thank you for the domain specific advices during the complete time of my thesis and your help in managing the last steps. Similarly, Martin Schäler and Veit Köppen were always present when problems occured, so that I could profit from their experiences. Thank you guys. Last but not least, I also express my thankfulness to the whole FeatureIDE team. Thus, thank you Thomas Leich for the successful collaborations during the improvement of the plug-ins and the interesting work. In this context, I also want to thank the guys of the METOP GmbH who were also responsible for the successful collaborations. However, the university part of FeatureIDE has similar importance. Thank you Thomas Thüm for managing the progress in FeatureIDE and I would like to thank all students who ensured that we

could achieve our goals. In this context, I particularly thank Sebastian Krieter for his great work in FeatureIDE and the support as student assistant. In addition, I also want to thank all other students, Jens Meinicke, Fabian Benduhn, Marcus Pinnecke, and all other I forgot in this place.

Besides the thankfulness to my research colleagues, I will never forget the support of my complete family. Now, it is time to say THANK YOU, since I now pass the finish line. Here, I want to particularly mention my wife Susann who supported me during the time of my bachelor, master and PhD thesis. Without her support I would not be here!

Foreword

To present the concept of multi-level interfaces, the thesis shares material with several papers. First of all, the conceptual idea of multi-level interfaces is described in the papers *Towards Modular Analysis of Multi Product Lines* [Schröter et al. 2013a] and *Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines* [Schröter 2014]. Furthermore, as the thesis presents insights into three interface levels, i.e., modeling, implementation, and behavior, the thesis also shares material with the corresponding papers. In detail, for the modeling level, the thesis shares material with the paper *Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems* [Schröter et al. 2016]. Considering the level of implementation, the thesis also shares material with the paper *Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines* [Schröter et al. 2014]. In addition, further shared material of the paper *Variability Hiding in Contracts for Dependent Software Product Lines* [Thüm et al. 2016] is used to describe the behavioral level and to give an overall example for the purpose of illustration. In sum, the thesis shares material with the following papers:

Schröter, R., Siegmund, N., and Thüm, T. (2013a). Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Software Product Line Conference co-located Workshops*, pages 96–99, New York, NY, USA. ACM.

Schröter, R., Siegmund, N., Thüm, T., and Saake, G. (2014). Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 102–111, New York, NY, USA. ACM.

Schröter, R. (2014). Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines. Technical Report 04, School of Computer Science, University of Magdeburg, Germany.

Schröter, R., Krieter, S., Thüm, T., Benduhn, F., and Saake, G. (2016). Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 667–678, New York, NY, USA. ACM.

Thüm, T., Winkelmann, T., Schröter, R., Hentschel, M., and Krüger, S. (2016). Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104, New York, NY, USA. ACM.

Contents

List of Figures

List of Tables

List of Acronyms

AOP Aspect-Oriented Programming

API Application Programming Interface

DOP Delta-Oriented Programming

FOP Feature-Oriented Programming

IDE Integrated Development Environment

JML Java Modeling Language

MPL Multi Product Line

OOP Object-Oriented Programming

SPL Software Product Line

UML Unified Modeling Language

1. Introduction

The reuse of software artifacts is a central part of the software-development process. It allows developers to create multiple software products in a more efficient manner compared to a product development from scratch. One popular type of software reuse is the mechanism *copy & paste* as it is the fastest way to take existing functionality and use it as base to customize the software to the developer's needs. This kind of software customization is also known as the clone-and-own approach [Rubin and Chechik 2013; Rubin et al. 2013]. Afterwards, as one option, we can use branches of a version control system to manage these similar products [Apel et al. 2013a]. Even if this mechanism produces fast results, it is an unstructured way to reuse software artifacts that also negatively affects the software maintainability [Apel et al. 2013a]. For instance, if an error exists in one code artifact, it needs to be fixed in all duplicates. Therefore, a structured procedure is needed to achieve a more efficient reuse of software artifacts. As a consequence, several implementation paradigms, such as Object-Oriented Programming (OOP), also focus on an improvement of software reuse [Meyer 1988]. OOP provides several mechanisms like inheritance to efficiently reuse existing functionality. In addition to OOP mechanisms, object-oriented design patterns were proposed to organize the source code in a manner so that duplicated code can be reduced [Gamma et al. 1995]. However, even with OOP and design patterns, it is difficult and time-consuming to create reusable artifacts and to tailor products to our needs. Again, the option to copy a product and to adapt specific parts of it is often faster than restructuring the code to enable efficient reuse.

Especially when we focus on implementing similar programs with alternative core implementations, the reuse of software artifacts is essential to reduce the development time and costs [Apel et al. 2013a; van der Linden et al. 2007]. However, even in this scenario, it is only an appropriate procedure to use mechanism like *copy & paste* iff we only focus on a few products. To overcome these limitations, software product lines (SPLs) can be used to structure the reuse of software artifacts so that we can implement similar programs on a common code base [Czarnecki and Eisenecker 2000]. In detail,

using software product lines we can achieve a tailored product based on a cost-reduced, quality-improved, and time efficient software-development process [Apel et al. 2013a]. Depending on the specific implementation strategy of the software product line, each characteristic can be more or less shaped.[1] However, all strategies of software product lines have in common that the systems are described based on features. As several definitions of a *feature* exist [Apel et al. 2013a; Kästner et al. 2011], it is necessary to clarify our understanding of it. We consider a feature as a characteristic of a software system that we use to describe the commonalities and differences of all products of a product line on the modeling, implementation and behavioral level. This viewpoint is similar to the definition of Apel et al., in which a feature is also used to manage the variation throughout the whole software lifecycle [Apel et al. 2013a].

The size and complexity of industrial systems is growing fast (e.g., considering the size of existing variability models in practice [Berger et al. 2013]), resulting in new challenges for product lines. For instance, the Linux kernel, a most famous example for a product line, consists of thousands of features (more than 11 000 features [Tartler et al. 2012]) with complex feature-dependencies. This complexity leads to challenges regarding the correctness of feature dependencies on the modeling, implementation and behavioral level. In this context, it is hard to find contradicting feature dependencies which can result in incorrect and undesired products. Even if automated analyses exist, which are able to detect inconsistencies (e.g., using sat-based analyses [Mendonça et al. 2009b]), the analyses suffer from the number of features. In particular, if an analysis is based on numerous satisfiability checks, a delay through every satisfiability check becomes significant. Unfortunately, the defined dependency between features and, thus, the satisfiability checks, are also needed in analyses of other development levels. For instance, it directly influence type checking (e.g.,[Czarnecki and Pietroszek 2006; Thaker et al. 2007]) for the implementation level as well as model checking (e.g., [Classen et al. 2010; Lauenroth et al. 2009]) when verifying the behavioral level. Another example that directly suffers from the complexity, is the implementation task, in which it is more difficult for a developer to find reusable code-artifacts from other features. Thus, it is difficult to avoid dangling references during the implementation of a product line.

A general concept to solve huge and highly complex problems is divide and conquer. This means, the problem is divided into several pieces that are easier to solve, and the partial solutions are combined to solve the complex problem. In the context of product lines this concept complies with the approach of multi product lines (MPLs), which represents an arbitrary composition of a set of product lines [Rosenmüller and Siegmund 2010]. According to another definition of multi product lines by Holl et al., a multi product line is a set of interdependent product lines. These product lines are still self-contained but, at the same time, the product lines can be used to describe (ultra) large-scale systems [Holl et al. 2012]. Hence, it is possible to decentralize the

[1] For an overview of classic and advanced techniques for the development of software product lines, we refer the reader to the book of Apel et al. that summarizes the advantages and drawbacks of each implementation strategy [Apel et al. 2013a].

development of the interdependent product lines of multi product lines and, thus, to reuse already existing (parts of) product lines to develop (ultra) large-scale systems. In particular, when looking at current trends of a software ecosystems the decentralization becomes even more important. In detail, software ecosystems focus on a development beyond the organizational boundaries [Bosch 2009; Bosch and Bosch-Sijtsema 2010; Galindo et al. 2015]. As a result, it is possible to further decrease the development time and costs [Bosch 2009].

Even if the concept of multi product lines is promising, the development of a multi product line is still a difficult task. Similar to product lines, we can divide the development process of multi product lines into several development levels that depend on each other (e.g., the modeling level defines the features to implement). However, there is also a close dependency between the involved product lines on each level. For instance, taking a deeper look into the modeling level, we have to define the interdependencies between the product lines of the multi product line. Afterwards, we need to analyze whether the model represents our intended concept. For instance, we can check if an assumption holds that two features always occur together (e.g., using techniques for automated analysis [Benavides et al. 2010]). However, even if we only plan to reuse a couple of features from another product line (i.e., features of interest), we have to combine the models with all features and we need to analyze the whole model of the multi product line [Schröter et al. 2013b]. Furthermore, if changes occur in feature dependencies, independent whether a feature is of our interest, we need to repeat the process. Similar dependencies exist on the other levels, such as the reuse of implementation artifacts (e.g., members of the Application Programming Interface (API)) from another product line. Because of the dependencies between the involved product lines, we need to know several details of the reused product line to find appropriate implementation artifacts for reuse. On top of the direct dependencies between the involved product lines, dependencies between the development levels additionally compound the problem. For instance, changes in the reused model can have an effect on the implementation artifacts that we reused on the implementation level. Thus, it is desirable to avoid the direct dependencies between the involved product lines of a multi product line especially when looking at software ecosystems with the focus on a development beyond organizational boundaries.

In this thesis, we propose multi-level interfaces as a concept to avoid direct dependencies between the involved product lines on each development level of a multi product line. Using multi-level interfaces, we aim to improve the analysis and evolution of the multi product line on each development level. For this purpose, we want to introduce interfaces for the modeling, implementation and behavioral level of a multi product line. The multi-level interfaces should be hierarchical, that is, interfaces on lower levels depend on the interfaces on the upper levels. As a result, we aim to get a holistic view on a product line, in which the details of a specific development level are hidden from the user's perspective. Thus, based on this thesis, we want to answer the following, global research questions to achieve our goals:

GRQ1: How can we represent the interfaces of the development levels of a multi product line?
For each interface of our multi-level interfaces regarding the modeling, implementation and behavioral level it is necessary to find a suitable representation of the specific interface so that also the dependencies to the upper levels are considered.

GRQ2: Is it possible to generate the interfaces of the development levels of a multi product line?
If we found a suitable representation for each interface, the question arises whether it is possible to generate the specific interface. Otherwise, manual effort is needed to create the interface, which reduces possible advantages of the interface.

GRQ3: Can we use multi-level interfaces to improve the analysis and/or evolution of multi product lines?
Based on the proposed interfaces, we want to investigate the benefits when using multi-level interfaces. Therefore, we need to take a look at each interface in detail.

To answer our research questions, we present an overview of our concept of multi-level interfaces with detailed goals for each interface level (cf. Chapter 3). In this context, we also introduce one hypothesis for each interface that we want to investigate to answer our third research question. Afterwards, we consider each interface in detail. First, we refine our initial concept for the specific interface using our initial view that we proposed with the holistic concept of multi-level interfaces [Schröter 2014; Schröter et al. 2013a]. Second, the concept of the specific interface is integrated into an evaluation environment. For instance, for the concept on the modeling and implementation level, we use FeatureIDE as it represents an Integrated Development Environment (IDE) for product lines [Meinicke et al. 2017; Thüm et al. 2014b]. Third, we use the specific interface integration to evaluate the concept regarding the corresponding hypothesis. For this purpose, we mainly use quantitative evaluations that are partly based on real-world case studies to compare the specific analysis of our concept with the state-of-the-art techniques. Afterwards, we discuss the outcome and threats to validity for the specific interface. In sum, we make the following contribution:

⇒ We introduce an interface for the modeling level of multi product lines that we call a *feature-model interface*. Based on this, we show how to achieve advantages regarding feature-model analyses in case of evolution.
⇒ We introduce the *variable interface* as an interface for the implementation level of multi product lines. We show how we can improve the support for the implementation of a multi product line so that it is easier to identify reusable code artifacts.
⇒ We introduce different strategies to define an interface for the description of the behavioral level of a multi product line. Using these strategies as *behavioral product-line interface*, we show how we can reduce the effort for the product-line verification.

⇒ We illustrate the general concept of multi-level interfaces based on different levels that depend on each other. In detail, we show how the variable interface depends on the feature-model interface and that the behavioral product-line interface depends on the information given by both upper levels.

Thesis Outline

To present the concept of multi-level interfaces and to illustrate the outcome of this thesis, we decided to structure the thesis according to the different interface levels that are based on each other. Therefore, the core of the thesis is formed by three main chapters that present interfaces for the modeling, implementation and behavior level of the (multi) product-line development. In addition, in our background and overview chapter, we present necessary information for the subsequent main chapters and give insights to their relations. Finally, we present related work, give a conclusion, and an overview of future work. The rest of this thesis is structured as follows:

Chapter 2: This chapter presents basic background knowledge for this thesis. The chapter is divided in two main parts. First, we present background information about software product lines (see Section 2.1). Second, in Section 2.2, we reuse the knowledge about software product lines to describe multi product lines. In general, both sections take a look at concepts for modeling and implementation, whereas the section of software product lines also presents some insights of specification techniques.

Chapter 3: This chapter presents an overview of the ideas of multi-level interfaces. In contrast to the corresponding interface chapters, in which each interface is described in detail, this overview chapter focuses on the description of the dependencies between the interfaces. In addition, we present how to analyze the benefits when using our interface levels and we introduce our hypotheses that we use to investigate our interfaces in the subsequent chapters.

Chapter 4: Here, we introduce the feature-model interface as the first interface of our concept of multi-level interfaces. We start with a presentation of a formal description of this interface. Afterwards, we describe dependencies between model analyses, which are based on direct model dependencies and dependencies to the proposed interfaces. To investigate the benefits when using this interface for model analyses, we use a real-world application for the evaluation.

Chapter 5: As the second interface of multi-level interfaces, we introduce variable interfaces for the level of implementation of multi product lines. We show how to use this interface to support the implementation of software product lines and multi product lines. Furthermore, for the purpose of evaluation, we compare the results with state-of-the-art approaches that can be used to support the implementation of (multi) product lines.

Chapter 6: As a proof of concept for our multi-level interfaces, this chapter introduces the behavioral product-line interface. Here, we present different strategies to implement this interface and show how to use this interface to avoid direct depen-

dencies. Afterwards, we use these different strategies to compare the suitability as a behavioral product-line interface.

Chapter 7: We use this chapter to present related work for the overall concept. We also present related work for each interface and work that consider similar goals (e.g., information hiding).

Chapter 8: In this chapter, we present a summary and the conclusion for this thesis. In addition, we present research directions for future work that would have exceeded the scope of this thesis.

2. Background - How to Develop (Multi) Product Lines

In this chapter, we introduce fundamentals that are required to comprehend the subsequent chapters. As we investigate techniques to ease the development and evolution of multi product lines, also called dependent product lines, we start with an introduction of product lines that are used as basic concept. In detail, we give insights into the development cycle of product lines and present how we can model, implement, and verify a single product line. Afterwards, we take a look into multi product lines and present a straight-forward development of these dependent product lines. Indeed, we do not aim to give a complete overview of all existing techniques but concentrate on techniques that we need for the description of the subsequent chapters.

2.1 Development of Single Product Lines

In this section, we give an overview regarding the development of software product lines (called product lines for simplicity). In general, a product line is a set of products (also called variants) from one specific domain (i.e., the products are developed for a similar aim) that shares a set of common features developed on a common code base [Clements and Northrop 2001]. Even if the term feature is well known in the domain of software product lines, the meaning partly differs [Apel et al. 2013a; Kästner et al. 2011]. In detail, a feature is often considered as a coarse-grain concern that we want to include into several products of this specific domain. By contrast, a feature can also be a very fine-grained option for specific products. However, all the different views on features have in common that a customer can chose a set of combinable features to achieve a product customized to their specific needs. From the viewpoint of the developer, the concept of software product lines has also several advantages. For instance, as a software product line is based on a common code base, we can improve the time to market as well as for maintenance of similar products and, thus, we can save money [Pohl et al. 2005].

In the following, we give deeper insights into the development of product lines. In detail, in Section 2.1.1, we present the concept of feature models, which is the common used technique for the modeling of product lines. Furthermore, we present implementation techniques of software product lines using feature-oriented programming and preprocessors in Section 2.1.2. Afterwards, in Section 2.1.3, we present fundamentals according to the behavioral level and, thus, we show details on the specification and verification of products and product lines.

2.1.1 Variability Modeling of Product Lines

Variability modeling is used to define the commonalities and differences of the product line's products. For this purpose, several variability modeling approaches were proposed such as feature modeling [Kang et al. 1990], decision modeling [Schmid et al. 2011], and orthogonal variability modeling [Pohl et al. 2005]. Several advantages and drawbacks exists for all approaches, for instance, decision modeling focuses on the differences of the products whereas the feature modeling approach takes also the commonalities into account [Czarnecki et al. 2012]. However, as the feature-modeling approach is commonly used for the variability modeling [Berger et al. 2013; Chen and Ali Babar 2011], we also used the concept for our investigation. In the following, we present insights into this variability modeling approach.

As already described, *feature models* are commonly used to describe features and their dependencies of a product line to ensure valid products. Besides other representations, Kang et al. introduced a graphical representation of the feature model as tree structure with additional cross-tree constraints, called feature diagram, in which all features are represented as tree elements [Kang et al. 1990]. The tree structure itself represents first dependencies. For instance, the selection of a child feature forces that also the parent feature has to be included into the products of the product line. Furthermore, the root feature of a feature diagram is included in all products of the product line. However, the representation of feature diagrams in several works differs (cf. [Batory 2005; Czarnecki and Eisenecker 2000; Czarnecki et al. 2005; Czarnecki and Wąsowski 2007; Kang et al. 1990]) and, thus, it is essential to specify further dependencies so that the meaning and comprehension is unique between the reader and us (cf. by a legend). In this thesis, we use the representation of feature diagrams that is used in FeatureIDE [Meinicke et al. 2017; Thüm et al. 2014b], a plug-in for the development of software product lines. Thus, we allow the subsequent dependencies:

- **Mandatory:** A child feature is marked as mandatory if it is included in all products, in which the parent feature is included. In FeatureIDE, it is only allowed to use the mandatory dependency in *and groups*.
- **Optional:** A child feature is marked as optional if the user can freely decide whether the feature should be included in a product. In FeatureIDE, it is only allowed to use the optional dependency in *and groups*.

And Group: The *and group* allows us to define multiple child features in which all features are either mandatory or optional features.

Or Group: The *or group* allows us to define multiple child features whereas at least one feature needs to be included in a product if the parent feature is included.

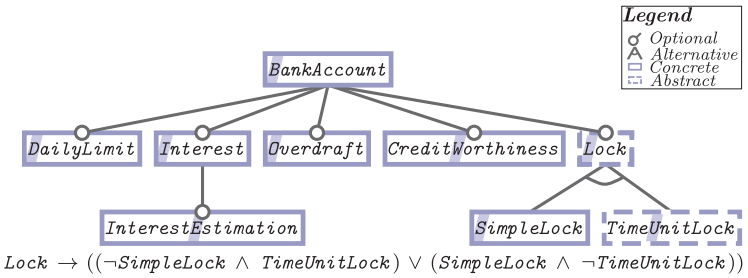
Alternative Group: The *alternative group* allows us to define multiple child features whereas exactly one child needs to be included in a product if the parent feature is included.

However, it is not possible in each scenario to represent all feature dependencies inside of this tree structure. Therefore, it is also possible to add further cross-tree constraints, in which we can define additional relations as arbitrary propositional formulas.

Feature Model by Example

In Figure 2.1(a), we present a small example of a feature model from the domain of bank software. In detail, we depict the feature model of a product line *BankAccount* that we use as running example allover the thesis. However, we use this example as it was also used in several other theses and papers [Thüm 2015; Thüm et al. 2014; Thüm et al. 2012] and it is a common case study for studies on product-line verification, which is also part of this thesis. To improve the illustration of our approaches for the concept of multi-level interfaces, we adapted the initial product line *BankAccount*. Therefore, we now introduce each feature and their relations and we reuse the product line to present our complete concept of multi-level interfaces.

The adapted product line *BankAccount* consists of nine features and, thus, we have to describe the dependencies between these features using the feature model. As described above, the root feature *BankAccount* is included in each product of the product line. Furthermore, the root feature has five optional subfeatures, *DailyLimit*, *Interest*, *Overdraft*, *CreditWorthiness*, and feature *Lock*. Therefore, the user can freely decide whether one of these features should be included in the final product. In this scenario, the feature *DailyLimit* should realize that a customer of this bank account can only withdraw a specific amount of money per day. Similarly, the feature *Overdraft* allows a customer to withdraw a specific amount of money even if this money is not available on the bank account. Furthermore, the feature *Interest* has to provide functionality to calculate the interest of this account and feature *CreditWorthiness* needs to check whether the costumer's behavior is conform to the credit conditions. Finally, the optional feature *Lock* can be used to provide lock functionality that controls the access to this account. However, if we choose feature *Lock* we also have to choose exactly one of its alternative subfeatures, feature *SimpleLock* or *TimeUnitLock*. Even if both features control the access to the account, the feature *SimpleLock* should provide an API to allow or deny the access, whereas the feature *TimeUnitLock* denies the access



(a) Feature diagram of product line *BankAccount*

$\mathcal{F}_{BankAccount} = \{$ B D I O W L E S U $\}$
$\mathcal{P}_{BankAccount} = \{\{$ B I O E $\}$
$\{$ B I O L E U $\}$
$\{$ B I O L E S $\}$
$\{$ B I O W E $\}$
$\{$ B I O W L E U $\}$
$\{$ B I O W L E S $\}$
$\{$ B D I O E $\}$
$\{$ B D I O L E U $\}$
$\{$ B D I O L E S $\}$
$\{$ B D I O W E $\}$
$\{$ B D I O W L E U $\}$
$\{$ B D I O W L E S $\}$
$\ldots$ $\}\}$

(b) Feature-model formalization for product line *BankAccount*

Figure 2.1: Feature model $\mathcal{M}_{BankAccount} = (\mathcal{F}_{BankAccount}, \mathcal{P}_{BankAccount})$ (highlighted characters of the feature diagram are used to represent $\mathcal{F}_{BankAccount}$ and $\mathcal{P}_{BankAccount}$).

for a specific amount of time. Because of the relations from the feature model, we also know that if we choose one of the subfeatures we also have to select the parent feature to achieve a valid product. This is also the case for our last feature of the product line *BankAccount*, called feature *InterestEstimation*. It is an optional subfeature of feature *Interest*, provides functionality to predict the interest of the next year, and forces that also the feature *Interest* and the root feature *BankAccount* needs to be included in a product if this feature is part of the product. Besides this relation inside of the tree structure, the feature model also provides a cross-tree constraint represented as propositional formula. Using the highlighted characters, the constraint can be represented as follows $L \rightarrow ((\neg S \wedge U) \vee (S \wedge \neg U))$. However, this constraint is identical to the dependencies given in the tree-structure of feature *Lock* and its subfeatures. Therefore, we only included this formula for the purpose of illustration of cross-tree constraints and, thus, we will remove it in all remaining representations.

The represented feature model with the described feature relations of product line *BankAccount* results in 72 different feature combinations, called configurations. However, some of the presented features are under construction or are only used to structure the feature model and, thus, do not contain real functionality. In detail, feature *Lock* does not contain functionality, as it is only used for structural reasons. Furthermore, the feature *TimeUnitLock* is under construction and needs further time for the functional completion. As a result, we marked both features as so called abstract features (cf. Figure 2.1, features with dashed border lines) [Thüm et al. 2011a]. By contrast, all other features are concrete features and provide real functionality. However, because of these abstract features, the real number of products that present different functionality is 48.

Formalization and Logical Representation of Feature-Models

So far, we used feature diagrams to represent the relations between features. However, it is also possible to list all valid feature combinations to present the set of products

or to use propositional formulas or textual representations to describe the relations in a feature model [Batory 2005]. In particular, the descriptions, proofs, and computations of this thesis are based on a special formalization and on the representation as propositional formulas.

Especially for our proof that we present in Chapter 4, we need to formalize a feature model. In detail, we use the set of all valid configurations and define a feature model as follows:

Definition 1. A feature model according to *Definition 1* of [Schröter et al. 2016]**:**

A feature model $\mathcal{M}_x$ is a tuple $(\mathcal{F}_x, \mathcal{P}_x)$, where
(a) $\mathcal{F}_x$ is a set of features, and
(b) $\mathcal{P}_x$ is a set of products with $\mathcal{P}_x \subseteq 2^{\mathcal{F}_x}$.

To exemplify our formalization, we use our running example of the product line *BankAccount*. In detail, we use the highlighted character of Figure 2.1 (a) in Figure 2.1 (b) to illustrate the definition of feature model $\mathcal{M}_{BankAccount}$ with the tuple $\mathcal{F}_{BankAccount}$ and $\mathcal{P}_{BankAccount}$. As the set of valid products is too large for the visual representation (i.e., 72 different feature combinations), we only present a subset in which we include all products that simultaneously consist of the features *Interest* (I), *InterestEstimation* (E), and *Overdraft* (O).

We can use the introduced formalization of feature models for our proofs regarding the dependencies of feature-model analysis results. However, for an evaluation or execution of feature-model analysis, this formalization is not suitable. By contrast, for the purpose of feature-model analyses, we use the feature-model representation as propositional formula. In this context, Batory described the dependencies between feature diagrams and their logic [Batory 2005] and Czarnecki and Wąsowski have shown that it is possible to transform the feature diagram into propositional formulas and back again [Czarnecki and Wąsowski 2007]. In the following, we use this knowledge to transfer feature diagram of Figure 2.1 into propositional logic.

$$\begin{aligned}
\text{Root Feature:} \quad & B && (1.1)\\
\text{Child-Parent:} \quad & \wedge D \rightarrow B \wedge I \rightarrow B \wedge O \rightarrow B \wedge W \rightarrow B \wedge L \rightarrow B && (1.2)\\
& \wedge E \rightarrow I \wedge S \rightarrow L \wedge U \rightarrow L && (1.3)\\
\text{Alternative Group:} \quad & \wedge L \rightarrow ((\neg S \wedge U) \vee (S \wedge \neg U)) && (1.4)\\
\text{Cross-Tree Constraint:} \quad & \wedge L \rightarrow ((\neg S \wedge U) \vee (S \wedge \neg U)) && (1.5)
\end{aligned}$$

The resulting propositional formula includes all feature relations of the tree structure and the additional cross-tree constraint of the feature model *BankAccount*. In detail, the formula describes the child-parent relation that forces the selection of the parent feature if a child feature is selected. Furthermore, we can see that the root feature

is included in all products and the necessity to select feature *SimpleLock* or feature *TimeUnitLock* if feature *Lock* is selected. Of course, it is also possible to transform all other relations of a tree-structure that we not used in our example (i.e., or groups). In detail, if an imaginary feature x is a mandatory subfeature of feature y, we will add the relation $y \rightarrow x$ so that we force the selection of x if y is selected. Furthermore, similar to the relation of an alternative group, we could add an or group. If we assume that the described alternative group is an or group, we would add $L \rightarrow (S \vee U)$ to this formula. However, besides the described transformation of the feature model's tree structure, we finally add all existing cross-tree constraints to this formula.

Even if we structured the propositional formula to improve the readability, it is not obvious which feature combinations fulfill the formula and, thus, lead to a valid product. However, as mentioned above, the purpose of the representation as propositional formula is to analyze the feature model to get advanced information or detect inconsistencies [Benavides et al. 2010]. As we want to improve the analysis of feature models in an evolution scenario of multi product lines, we now present different feature-model analyses and use our formalization of feature models to formally define them.

Feature-Model Analyses

To ensure the correctness of feature models, or to investigate whether the desired expressiveness and products exist, or to get advanced information, we need to analyze feature models. In particular, if a feature model consists of hundreds or thousands of features, like industrial systems show [Berger et al. 2013], feature-model analyses are essential. In this thesis, we present some proofs regarding the relation of analysis results for specific feature-model analyses (see Chapter 4 for more details). For this purpose, we formalize five of the commonly used analyses for feature models. However, for the application of an analysis, we also use the straight-forward concept. In detail, the feature model is transformed to a specific representation (e.g., propositional formulas that we described in the last paragraph) and a corresponding solver is used for the analysis execution [Benavides et al. 2010].

First of all, we consider the analysis of *void feature models*. A feature model is void iff it does not represent a product [Batory 2005; Benavides et al. 2010; Kang et al. 1990]. In detail, it exists a contradiction inside of the feature model so that no feature combination leads to a valid product. Therefore, the analysis is used to detect such contradictions as early as possible [Batory 2005; Benavides et al. 2010; Kang et al. 1990; Trinidad et al. 2008]. Here, we use our feature-model formalization (cf. Definition 1) to define the analysis of *void feature model*s. According to Definition 2, we assume that an universe of all feature models exists. As a result, we consider each feature model $\mathcal{M}_x$ of this universe $\mathcal{M}$ as *void* if the set of products $\mathcal{P}_x$ is empty. The result is a new set of feature models in which all feature models are void. Considering feature model *BankAccount*, we get the conclusion that this feature model is not part of all void feature models of the universe.

Definition 2. Analysis of *void feature models* according to *Definition 2* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{M}$ the set of all feature models of the universe, then

$$void = \{\mathcal{M}_x \in \mathcal{M} \mid \mathcal{P}_x = \emptyset\} \tag{2.1}$$

In our second formalization of feature-model analyses, we consider the analysis of *core features*. A core feature is a feature that is included in all products of a product line [Benavides et al. 2010; Trinidad and Ruiz-Cortés 2009]. In particular, the analysis is used to determine the features that we should implement first [Benavides et al. 2010; Trinidad et al. 2004]. This means, we can use the knowledge of core features to create an implementation priority list for all features. In Definition 3, we define a function *core* that uses the feature model $\mathcal{M}_x$ as input and determines the intersection of the sets of all products $\mathcal{P}_x$ to achieve the core features. Again, we consider our running example to illustrate the application of function *core*. If we use feature model $\mathcal{M}_{BankAccount}$ as input, we get the set $\{BankAccount\}$ of core features as result.

Definition 3. Analysis of *core features* according to *Definition 2* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{M}$ the set of all feature models of the universe, then

$$core(\mathcal{M}_x) = \bigcap_{p \in \mathcal{P}_x} p \tag{3.1}$$

The analysis of *dead features* represents our third formalization. In detail, if we consider a non-void feature model, a dead feature is a feature that is not included in any product of the specific product line [Benavides et al. 2010; Kang et al. 1990]. Therefore, similar to the analysis of void feature models, the analysis is used to detect contradictions in the feature model [Hemakumar 2008]. Thus, using this analysis, we don't waste time to implement a feature that we cannot use for any product line's product. In Definition 4, we define function *dead* with the feature model $\mathcal{M}_x$ as input to determine the dead features. In detail, the function determines the union of all features given in all products $\mathcal{P}_x$ and computes the difference to set of feature $\mathcal{F}_x$. If we use feature model *BankAccount* as input for function *dead*, we get an empty set as result. However, if we assume that the constraint $Lock \rightarrow SimpleLock$ as further cross-tree constraint exists, we achieve the set $\{TimeUnitLock\}$ as result. Assuming this additional dependency, it exists no product in which the feature *TimeUnitLock* is included.

Definition 4. Analysis of *dead features* according to *Definition 2* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{M}$ the set of all feature models of the universe, then

$$dead(\mathcal{M}_x) = \mathcal{F}_x \setminus \bigcup_{p \in \mathcal{P}_x} p \tag{4.1}$$

Our fourth formalization considers the analysis of partial configurations. First of all, a partial configuration of a feature model $\mathcal{M}_x$ is a tuple of selected feature $\mathcal{F}_S$ and deselected feature $\mathcal{F}_D$, whereas $\mathcal{F}_S \cup \mathcal{F}_D \subseteq \mathcal{F}_x$ and $\mathcal{F}_S \cap \mathcal{F}_D = \emptyset$. By contrast, we consider a configuration, in which additionally the union of selected and deselected features results in the set of all features, as full configuration (i.e., $\mathcal{F}_S \cup \mathcal{F}_D = \mathcal{F}_x$) [Benavides et al. 2010]. Based on this definition, the analysis of partial configurations determines whether a partial configuration fulfills all relations of the corresponding feature model [Batory 2005; Benavides et al. 2010; Kang et al. 1990]. In Definition 5, we define a function *pConf* that determines for a given feature model $\mathcal{M}_x$ the set of all tuples of selected and deselected features that fulfill the relations of features given by the feature model. To illustrate the application of function *pConf*, we use our feature model *BankAccount*. As a result, we can conclude that the configuration $(\{B\}, \{D, I, O, W, L, E, S, U\})$ is a valid full configuration of the feature model $\mathcal{M}_{BankAccount}$ as this tuple is part of the result set and, thus, fulfills all feature-model relations. In addition, the configuration $(\{L\}, \{S\})$ is also a valid configuration and part of the result set of function *pConf*. However, by contrast to our first example, this is only a partial configuration. Furthermore, as the configuration $(\{B, S\}, \{L\})$ contains a contradiction, the configuration is not part of the result set of function *pConf* with feature model $\mathcal{M}_{BankAccount}$ as input.

Definition 5. Analysis of *partial configurations* according to *Definition 2* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{M}$ the set of all feature models of the universe, then

$$pConf(\mathcal{M}_x) = \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists p \in \mathcal{P}_x : \mathcal{F}_S \subseteq p \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_x \setminus p\} \tag{5.1}$$

Finally, we consider the formalization of the analysis *atomic sets*. In detail, an atomic set is a non-empty set of features that is completely included or completely absent in all product line's products. Therefore, we can replace the features of an atomic set by one single feature of this set (e.g., considering a propositional formula) and, thus, atomic sets can reduce the complexity of other feature-model analyses [Benavides et al. 2010; Segura 2008; Zhang et al. 2004]. However, the first idea of a set of features that can be considered as unit, is based on a mandatory parent-child dependency [Benavides et al. 2010; Zhang et al. 2004]. Therefore, several implementations of an atomic set analysis only consider this kind of dependencies. By contrast, similar to Durán et al. [Durán et al. 2017], we consider a set of features as an atomic set if our initial condition holds, in

which the set is always completely included in a product or completely absent. Because of cross-tree constraints, this can also result in an atomic set of features that are far away from each other considering the tree structure. Therefore, in Definition 6, we formalize the analysis of atomic sets using function *aSet* that uses a feature model $\mathcal{M}_x$ as input and returns a set of atomic feature sets as output (this formalization is similar to the definition of atomic sets given by [Durán et al. 2017]). To ease the definition of *aSet*, we use a second function *aSub* that determines all atomic subsets of a feature model $\mathcal{M}_x$. In detail, the result of function *aSub* is a set of feature sets whereas a set of features can be simultaneously a subset of another set. Therefore, the function *aSet* removes all these subsets from the result of function *aSub* so that only atomic supersets are presented as result. For instance, considering our feature model *BankAccount*, we get the result, that only sets of single features are returned as atomic sets. However, if we again assume an additional cross-tree constraint $Lock \rightarrow SimpleLock$, we get another result. Here, the function *aSub* presents the set $\{\{L, S\}, \{L\}, \{S\}, ...\}$ as output. After the application of function *aSet*, only the superset $\{L, S\}$ and the single sets of the other features still remain.

Definition 6. Analysis of *atomic sets* according to *Definition 2* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{M}$ the set of all feature models of the universe, then

$$aSet(\mathcal{M}_x) = \{p \in aSub(\mathcal{M}_x) \mid \forall q \in aSub(\mathcal{M}_x) : p \not\subset q\} \tag{6.1}$$

$$aSub(\mathcal{M}_x) = \{q \mid q \neq \emptyset, \mathcal{P}_x \neq \emptyset, \forall p \in \mathcal{P}_x : q \subseteq p \vee q \subseteq \mathcal{F}_x \setminus p\} \tag{6.2}$$

Even if it is not part of our formalization but used in several subsequent descriptions, we have to consider the analysis of *false-optional features*. However, several definitions of a *false-optional feature* exist. For instance, considering the description of Benavides et al., a *false-optional feature* is a feature that is included in all products of the product line even if it is not modeled as a mandatory feature [Benavides et al. 2010]. Using this definition, the set of all *false-optional features* of a product line is a subset of all *core features* of the same product line. By contrast, we consider a *false-optional feature* as a feature that is modeled as optional in the feature diagram but it also has a mandatory relation to the parent feature [Meinicke et al. 2017]. As result, the *false-optional feature* is not automatically a *core feature*. However, both definitions have in common that a specific feature combination without the *false-optional feature* but with the parent feature is indicated even if this feature combination is not possible. Therefore, the analysis is used to avoid such problems. As our feature-model formalization does not include information about the graphical structure of the feature model (i.e., the feature diagram), it is not possible to formalize this analysis based on it. Nevertheless, it is possible to determine the *false-optional features* of a feature model using the representation as propositional formula. Using this representation and our running example of product line *BankAccount*, we get an empty set of *false-option features* as result. However, if we assume that the constraint $Overdraft \rightarrow InterestEstimation$ exists, we get a set with the features *Interest* and *InterestEstimation* as result.

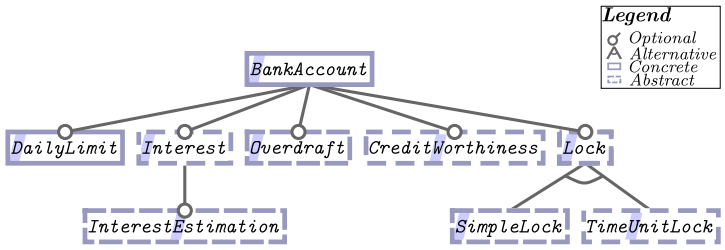


Figure 2.2: Feature model *BankAccount* that represents two products, a product with feature *BankAccount* and a second product with the features *BankAccount* and *DailyLimit*.

2.1.2 Implementation of Product Lines

Based on the feature models given in the last section, we know the features and their dependencies that we want to implement. For the implementation itself, we have the choice between several composition-based and annotation-based techniques. Whereas composition-based techniques modularize the feature code, the annotation-based techniques map code artifacts to the features (i.e., using ifdef directives) [Apel et al. 2013a]. Even if we exemplify our concept using the composition-based technique feature-oriented programming, we also briefly introduce annotation-based techniques as we also want to discuss their application with our concept. For the illustration of the different implementation techniques, we use an excerpt of our running example of product line *BankAccount*.

In Figure 2.2, we present a slightly different feature model of our product line *BankAccount*, in which we marked all features as abstract that we plan to implement later. As result, we start with the implementation of the features *BankAccount* and *DailyLimit*. Therefore, the current feature model allows us to create two products one with feature *BankAccount* and one with feature *BankAccount* and feature *DailyLimit*. We depict the source code of both products in Figure 2.3.

In the following, we present different implementation techniques that allow us to generate products with the same behavior as given in Figure 2.3. As preprocessors are commonly known in industry, we start with an implementation based on the preprocessor Antenna [Pleumann et al. 2011]. Afterwards, we introduce feature-oriented programming that is used in our running example for the concept of multi-level interface.

Preprocessor

In this section, we illustrate the application of preprocessors for the implementation of software product lines. For the concept of preprocessors, a developer maps variable code artifacts to features using annotations (i.e., #ifdef *feature*). In general, the annotations can consist of arbitrary propositional formulas. However, often the annotation corresponds to a specific feature or the negation of it. In a preprocessing step of the

```
1 public class Account {
2   int balance = 0;
3
4   boolean update(int x) {
5     int newBalance = balance + x;
6     if (newBalance < getOverdraftLimit())
7       return false;
8     balance = newBalance;
9     return true;
10  }
11
12  int getOverdraftLimit() {return 0;}
13 }
```

(a) Product based on feature *BankAccount*

```
14 public class Account {
15   int balance = 0;
16   final static int DAILY_LIMIT = -1000;
17   int withdraw = 0;
18
19   boolean update(int x) {
20     int newWithdraw = withdraw;
21     if (x < 0)  {
22       newWithdraw += x;
23       if (newWithdraw < DAILY_LIMIT)
24         return false;
25     }
26
27     int newBalance = balance + x;
28     if (newBalance < getOverdraftLimit())
29       return false;
30     balance = newBalance;
31
32     withdraw = newWithdraw;
33     return true;
34   }
35
36   int getOverdraftLimit() {return 0;}
37 }
```

(b) Product based on feature *BankAccount* and feature *DailyLimit*

Figure 2.3: Two products of the product line *BankAccount*.

```
1 public class Account {
2   int balance = 0;
3   //#if DailyLimit
4 //@ final static int DAILY_LIMIT = -1000;
5 //@ int withdraw = 0;
6   //#endif
7
8   boolean update(int x) {
9     //#if DailyLimit
10//@   int newWithdraw = withdraw;
11//@   if (x < 0)  {
12//@     newWithdraw += x;
13//@     if (newWithdraw < DAILY_LIMIT)
14//@       return false;
15//@   }
16    //#endif
17
18    int newBalance = balance + x;
19    if (newBalance < getOverdraftLimit())
20      return false;
21    balance = newBalance;
22
23    //#if DailyLimit
24//@   withdraw = newWithdraw;
25    //#endif
26    return true;
27  }
28
29  int getOverdraftLimit() {return 0;}
30 }
```

Figure 2.4: Preprocessor-based implementation of product line *BankAccount* using Antenna.

compiler, the preprocessor analyzes the annotations and removes code artifacts that lead to a formula with a `false` statement. As a result, the final compiled program only consists of code artifacts that lead to a `true` statement and, thus, the products only include features of our interest.

As our running example of product line *BankAccount* is based on Java, we use the preprocessor Antenna [Pleumann et al. 2011] for the illustration of the concept. Therefore, in Figure 2.4, we start with the code of our second *BankAccount* product (cf. Figure 2.3 (b)) and add preprocessor annotations to mark the code corresponding to feature *DailyLimit* as variable. In detail, we create an #ifdef annotation (cf. Line 3) with feature *DailyLimit*, which marks that the variable code begins on this position. Furthermore, we set an #endif annotation to finalize the variable block (cf. Line 6). Afterwards, we create two other blocks that surround the source code of the feature *DailyLimit* (cf. Lines 9–16 and Lines 23–25). If we then apply the preprocessor Antenna without feature *DailyLimit*, the source code belonging to this feature is automatically converted to a Java comment. As a result, the Java compiler only considers the uncommented source code.

Feature-Oriented Programming

By contrast to preprocessors, feature-oriented programming is a composition-based technique that allows us to separate the source code of features in dedicated feature modules [Batory et al. 2004; Prehofer 1997]. In detail, we can define several classes with members (e.g., methods, fields) in a feature module that corresponds to a specific feature. During the generation of a product, we can use the approach of superimposition [Apel et al. 2009, 2013b]. In detail, this means that all classes with identical names are recursively superimposed so that the final class contains all members of all input classes. If a member consists in several feature modules, rules exist to decide how to superimpose the content. For instance, we can use special keywords in methods to call the identical method defined in another feature module (e.g., keyword `original` in FeatureHouse [Apel et al. 2009, 2013b]).

To exemplify the concept, we use FeatureHouse, a language-independent composer for feature-oriented programming [Apel et al. 2009, 2013b], to implement product line *BankAccount*. In Figure 2.5, we depict the feature modules *BankAccount* and *DailyLimit*. In detail, the feature module *BankAccount* defines the class `Account` with the field `balance` and the methods `update` and `getOverdraftLimit`. The feature module *DailyLimit* defines the final field `DAILY_LIMIT` and the field `withdraw`. However, the feature module also defines the method `update` as a refinement of other feature modules. In our scenario, the method `update` of feature module *DailyLimit* refines the method `update` of feature module *BankAccount* and, thus, calls the initial functionality of this function using the keyword `original` (cf. Line 25).

As we use feature-oriented programming to exemplify our concept of multi-level interfaces, we also need a formalization of the product's generation process (i.e., of a specific product-line's product). For the feature-module composition, we need to consider all

```
1 public class Account {
2   int balance = 0;
3
4   boolean update(int x) {
5     int newBalance = balance + x;
6     if (newBalance < getOverdraftLimit())
7       return false;
8     balance = newBalance;
9     return true;
10  }
11
12  int getOverdraftLimit() {return 0;}
13 }
```

(a) Feature module of feature *BankAccount*

```
14 class Account {
15   final static int DAILY_LIMIT = -1000;
16   int withdraw = 0;
17
18   boolean update(int x) {
19     int newWithdraw = withdraw;
20     if (x < 0)  {
21       newWithdraw += x;
22       if (newWithdraw < DAILY_LIMIT)
23         return false;
24     }
25     if (!original(x))
26       return false;
27     withdraw = newWithdraw;
28     return true;
29   }
30 }
```

(b) Feature module of feature *DailyLimit*

Figure 2.5: Implementation of product line *BankAccount* using feature-oriented programming.

implementation units according to the selected features $\mathcal{F}_S$ of a specific product $p \in \mathcal{P}$. For this purpose, we adapt the definition of Apel et al. [Apel et al. 2013c] and define the composition of the implementation units as given in the Definition 7.

Definition 7. Feature-Module Composition (adapted definition of [Apel et al. 2013c])**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model, $f_1 ... f_n \in \mathcal{F}_S$ all selected features of a specific product $p \in \mathcal{P}_x$, $impl(f)$ the implementation unit of the feature f, and $impl(p)$ the composition of product p. Furthermore, $\bullet$ is the composition operator with $I \times I \rightarrow I$ defined over the set of all implementation units. We define the feature-module composition as follows:

$$impl(p) = impl(f_1) \bullet ... \bullet impl(f_n)$$

2.1.3 Specification and Verification of Product Lines

To investigate whether our concept of multi-level interfaces can also be applied to further concepts than modeling and implementation, we want to make a proof of concept on a further level. In detail, we decide to also investigate specification and verification techniques for product-lines that are used to ensure the correctness of product-line's products. For this purpose, we now introduce basics for the specification and verification of object-oriented programs. Afterwards, we show how to use these techniques in the context of product lines.

Program Verification in General

To create reliable object-oriented programs, we can use design by contract [Meyer 1988, 1992]. Based on design by contract, it is possible to assign preconditions (i.e., `requires`

```
1class Account {
2  //@ invariant balance >= getOverdraftLimit();
3  int balance = 0;
4  //@ ensures balance == 0;
5  Account() { }
6  //@ requires x != 0;
7  //@ ensures (\result ==> balance == \old(balance) + x) &&
8  //@         (!\result ==> balance == \old(balance));
9  boolean update(int x) {
10   if (balance + x < getOverdraftLimit())
11     return false;
12   balance += x;
13   return true;
14 }
15 //@ ensures \result == 0;
16 /*@ pure @*/ int getOverdraftLimit() {return 0;}
17}
```

Figure 2.6: Product *BankAccount* based on feature *BankAccount* with JML specifications.

clauses) and postconditions (i.e., `ensures` clauses) to methods. In detail, these conditions define valid states of the program, whereas the precondition needs to hold before and the postcondition after the method is executed [Hatcliff et al. 2012]. Furthermore, it is also possible to define invariants that additionally define the state of objects and classes. These are global constraints that also need to be hold when methods are executed or fields are changed. Based on such a specification, it is possible to use different kinds of languages and tools to verify the program correctness. In Java for instance, we can use the Java Modeling Language (JML) to specify the programs behavior. Using deductive verification, the Java code and the annotations of JML are translated into a specific logic that is subsequently used to prove that the program behaves according to the specification for each input value [Beckert et al. 2007].

In Figure 2.6, we depict a bank account product in which only feature *BankAccount* is included. Using JML, we specify the behavior of this product. In detail, we add the postcondition `ensures balance == 0` to the constructor of class `Account` so that it is ensured that the field `balance` is initialized with `0` (cf. Line 4). Furthermore, for the purpose of illustration, we add a precondition to method `update` so that the input value needs to be dissimilar to `0` (cf. Line 6). In addition, the postcondition of method `update` ensures that the account's balance is changed to the input if the method returns `true` and otherwise that the balance is constant (cf. Lines 7–8). The postcondition of method `getOverdraftLimit()` simply ensures that the method returns 0 in all executions (cf. Line 15). The additional keyword *pure* illustrates that the method is side-effect free [Beckert et al. 2007] and, thus, it allows us to use this method in specifications (cf. Line 16). This is necessary as we use this method in the invariant `balance >= getOverdraftLimit()` in which we define that the balance has to be greater or equal to this overdraft limit during all state changes (cf. Line 2). After the specification of the product's behavior, we can use concepts and tools for deductive verification, such as the theorem prover KeY [Beckert et al. 2007].

```
1 class Account {
2   //@ ensures \result == -5000;
3   /*@ pure @*/ int getOverdraftLimit(){ return -5000; }
4 }
```

```
5 class Account {
6   //@ invariant withdraw >= DAILY_LIMIT;
7   final static int DAILY_LIMIT = -1000;
8   int withdraw = 0;
9   //@ requires \original;
10  //@ ensures \original;
11  //@ ensures !\result ==> withdraw == \old(withdraw);
12  //@ ensures \result ==> withdraw <= \old(withdraw);
13  boolean update(int x) {
14    int newWithdraw = withdraw;
15    if (x < 0)  {
16      newWithdraw += x;
17      if (newWithdraw < DAILY_LIMIT) return false;
18    }
19    if (!original(x)) return false;
20    withdraw = newWithdraw;
21    return true;
22  }
23 }
```

Figure 2.7: Feature modules *Overdraft* and *DailyLimit* with JML specifications.

Program Verification for Product-Line's Products

In general, it is possible to create a product-line's product and to specify and verify this product afterwards. However, as a product line can result in thousands or millions of products, this task is cumbersome and error-prone as many specifications will be identical. Therefore, additional concepts are needed to specify and verify product lines in an efficient manner. In this context, we consider *feature-oriented contracts* that were introduced for the paradigm of feature-oriented programming and allow us to generate the specification similar to the source code [Thüm 2015]. However, the concept only reduces the specification effort for each product and, thus, we need an additional technique to reduce the verification effort of the whole product line. For this purpose, *variability encoding* can be used [von Rhein et al. 2016]. In the following, we introduce both techniques and use the product line *BankAccount* as running example for the purpose of illustration.

Feature-oriented contracts are contracts introduced for the paradigm of feature-oriented programming that allow us to compose a product specification similar to the composition of source code [Thüm 2015; Thüm et al. 2012]. In detail, feature-oriented contracts are contracts for feature modules represented as an extended version of design by contract that allows a developer to refine existing specifications of other feature modules (e.g., using the keyword `original`). Using the partial specification as input, we can generate a specific product specification during the product composition. The result is a product with a specification based on a common specification language that can be used as input for state-of-the-art tools for deductive verification, such as KeY [Beckert et al. 2007].

```
1class Account {
2  //@ invariant balance >= getOverdraftLimit();
3  int balance = 0;
4  //@ invariant withdraw >= DAILY_LIMIT;
5  final static int DAILY_LIMIT = -1000;
6  int withdraw = 0;
7
8  //@ ensures balance == 0;
9  Account() { }
10
11 //@ requires x != 0;
12 //@ ensures (\result ==> balance == \old(balance) + x) &&
13 //@         (!\result ==> balance == \old(balance));
14 boolean update_BankAccount(int x) {
15   if (balance + x < getOverdraftLimit())
16     return false;
17   balance += x;
18   return true;
19 }
20
21 //@ requires x != 0;
22 //@ ensures (!\result ==> balance == \old(balance)) &&
23 //@         (\result ==> balance == \old(balance) + x);
24 //@ ensures (!\result ==> withdraw == \old(withdraw)) &&
25 //@         (\result ==> withdraw <= \old(withdraw));
26 boolean update(int x) {
27   int newWithdraw = withdraw;
28   if (x < 0) {
29     newWithdraw += x;
30     if (newWithdraw < DAILY_LIMIT) return false;
31   }
32   if (!update_BankAccount(x)) return false;
33   withdraw = newWithdraw;
34   return true;
35 }
36
37 //@ ensures \result == -5000;
38 /*@ pure @*/ int getOverdraftLimit() { return -5000; }
39}
```

Figure 2.8: Generated source code with JML specifications for product *BankAccount* based on the features *BankAccount*, *DailyLimit*, and *Overdraft*.

To illustrate the concept of feature-oriented contracts, we use the product line *BankAccount* that we specified by the extended version of JML [Thüm 2015]. By contrast to the previous examples, we use a product-line version, in which also the feature *Overdraft* is implemented (cf. concrete features of Figure 2.9). In Figure 2.7, we present the feature modules *DailyLimit*, and *Overdraft* with feature-oriented contracts. As the contracts of the feature module *BankAccount* are identical to product *BankAccount* depicted in Figure 2.6, we focus on the other feature modules. In feature module *Overdraft*, we completely override the contract of feature module *BankAccount* and define a postcondition, in which we ensure the return value of `-5000` (cf. Line 2). By contrast, in feature module *DailyLimit*, we refine the JML postcondition of method `update`. In detail, we use the keyword `original` to refer to the contract definition of feature module *BankAccount* (cf. Lines 9–10). In addition, we add further constraints so that the state of field `withdraw` is lower or equal than the state before the method

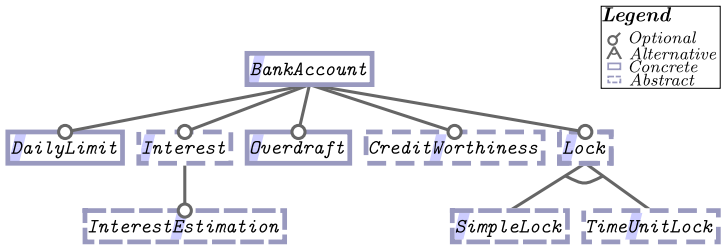


Figure 2.9: Feature model *BankAccount* that represents four products, one product with feature *BankAccount* and three other products in which also the features *DailyLimit* or/and *Overdraft* are included.

was executed, otherwise the contract ensures that the state of the field is constant (cf. Lines 11–12). Furthermore, we define another invariant that ensures that the state of the field `withdraw` is never below the daily limit that is allowed to withdraw (cf. Line 6). Based on this common code base with feature-oriented contracts, we can generate a *BankAccount* product with plain JML contracts. In Figure 2.8, we depict the *BankAccount* product with the features *BankAccount* and *DailyLimit*. Here, the keyword `original` was replaced by the contract of feature module *BankAccount* (cf. Lines 21–25). Thus, the contracts of method `update` consist of all incoming contracts of all feature modules and can be verified in a state-of-the-art fashion.

As illustrated above, feature-oriented contracts can be used to specify a product-line's behavior and to generate plain JML annotations that can be used for the product's verification. However, the previous concept does not reduce the verification effort as each product has to be verified separately. By contrast, a desired verification strategy is able to verify the complete product line without an additional verification of each product. For this purpose, we can use variability encoding that can also be applied to feature-oriented contracts. In detail, variability encoding is used to transform the compile-time variability into run-time variability [von Rhein et al. 2016]. As a result, the product line is represented in one product called metaproduct that includes the functionality of all product-line's products. To control the behavior of this metaproduct, the metaproduct introduces new fields that represent the features of the feature model which are used to define branches with different behavior according to the feature definitions. At the same time, we can also apply variability encoding to feature-oriented contracts to encode the variability of specifications into this metaproduct [Thüm et al. 2014; Thüm et al. 2012]. This allows us to use state-of-the-art tool support for product verifications. As a result, we can verify all products of the product line in one step.

For the purpose of illustration, we apply variability encoding to our product line *BankAccount* (cf. Figure 2.9) and present the results in Figure 2.10. For each concrete feature, we introduce a field in the new created class `FM` that represents the product-line's features (cf. Line 2). The values of these fields are used to control the run-time behavior according to a specific product of the product line. For instance, method

```
1 class FM {
2   static boolean bankAccount, dailyLimit, overdraft;
3 }
```

```
4 class Account {
5   //@ invariant FM.bankAccount && (!FM.dailyLimit || FM.bankAccount) &&
6   //@ (!FM.overdraft || FM.bankAccount);
7
8   //@ invariant balance >= getOverdraftLimit();
9   int balance = 0;
10
11  //@ ensures balance == 0;
12  Account() { }
13
14  //@ invariant FM.dailyLimit ==> withdraw >= DAILY_LIMIT;
15  final static int DAILY_LIMIT = -1000;
16  int withdraw = 0;
17
18  //@ requires x != 0;
19  //@ ensures (!\result ==> balance == \old(balance)) &&
20  //@         (\result ==> balance == \old(balance) + x);
21  //@ ensures FM.dailyLimit ==> ((!\result ==> withdraw == \old(withdraw)) &&
22  //@                          (\result ==> withdraw <= \old(withdraw)));
23  boolean update(int x) {
24    if (!FM.dailyLimit) return update_BankAccount(x);
25    int newWithdraw = withdraw;
26    if (x < 0)  {
27      newWithdraw += x;
28      if (newWithdraw < DAILY_LIMIT) return false;
29    }
30    if (!update_BankAccount(x)) return false;
31    withdraw = newWithdraw;
32    return true;
33  }
34  boolean update_BankAccount(int x) { /*[...]*/ }
35
36  //@ ensures (!FM.overdraft ==> \result == 0);
37  //@ ensures (FM.overdraft ==> \result == -5000);
38  int /*@ pure @*/ getOverdraftLimit(){
39    if (!FM.overdraft) return 0;
40    return -5000;
41  }
42 }
```

Figure 2.10: Variability encoding for class `Account` of product line *BankAccount*.

`getOverdraftLimit` was initially introduced in the feature modules *BankAccount* and *Overdraft*. Variability encoding either combines method definition of feature modules by inlining or it renames methods so that they can be referred by the other methods with initially the same name. For method `getOverdraftLimit`, inlining is used (cf. Lines 38–41). Thus, the method returns `0` if the field `overdraft` of feature model `FM` is `false` (cf. Line 39), or it returns `-5000` otherwise (cf. Line 40). By contrast, for the variability encoding of method `update`, the method of feature module *BankAccount* is represented in a method named `update_BankAccount`. Thus, depending on the value of field `dailyLimit`, the method `update` returns the result of method `update` without feature *DailyLimit* (cf. Line 24), or the result based on the features *DailyLimit* and *BankAccount* (Lines 25–32). Similar to the variable method definitions, we handle

variable fields. In detail, if a field is instantiated with different values in the feature modules, the initialization in the metaproduct depends on a variable of class `FM`.

If we take a look at the variable encoding of the contracts, the concept is similar to the source code encoding. First of all, an additional invariant is created in which all valid feature combinations are described using a logical representation of the feature model (cf. Lines 5–6). This ensures that we only prove valid products and that invalid products do not affect the results in a negative fashion. Furthermore, variability encoding combines variable contracts using dependencies to the feature variables of class `FM`. For instance, in the postcondition of method `update`, the keyword `original` is replaced by the postcondition of feature module *BankAccount* (cf. Lines 19–20), whereas the optional postcondition of feature module *DailyLimit* depends on the variable `dailyLimit` of class `FM` (cf. Lines 21–22). As method `getOverdraftLimit` uses overriding, the postcondition depends on the field `overdraft` (cf. Lines 36–37). If the state of the field is `true`, the contract ensures a value of `-5000` as return value, otherwise it ensures the value `0`. Similarly, variability encoding handles invariants. In detail, as the invariant `withdraw >= DAILY_LIMIT` should only be available if feature *DailyLimit* is active, the invariant in the metaproduct depends on the field `dailyLimit` of class `FM` (cf. Line 14). In sum, using the resulting metaproduct including the feature-oriented contracts, it is possible to verify the complete product line including all represented products in one verification process.

2.2 Development of Multi Product Lines

In this section, we introduce the concept of *multi product lines*, which are also called *dependent product lines* [Rosenmüller et al. 2008; Thüm et al. 2016]. Whereas the idea of product lines is to optimize the reuse of software artifacts from one specific domain, multi product lines optimize the reuse of already existing product lines, partly from multiple domains, in one huge variable system. Therefore, Holl et al. define a multi product line as a set of self-contained product lines with interdependencies that together describe a large-scale system [Holl et al. 2012]. As a result, it is also possible to decentralize the development of the underlying product lines. In particular, considering the concept of software ecosystems, the development can also be realized beyond the organizational boundaries to further reduce the development time and costs [Bosch 2009; Bosch and Bosch-Sijtsema 2010]. From another viewpoint, the concept of multi product lines tackles the reuse problem that no single product of a product line fulfills at the same time all requirements for a reuse in different products of another product line. Therefore, the selection of features in one product line can force a specific selection of features in another product line [Galindo et al. 2015]. As a result, the multi product line can have complex dependencies that we have to describe through the modeling, consider in the implementation, and specification of the multi product line.

In the following, we take a look on variability modeling in the context of multi product lines and focus on the specific concepts on which this thesis is based on. Afterwards, we give insights on how to implement a multi product line.

2.2.1 Variability Modeling of Multi Product Lines

As mentioned above, a multi product line is a set of software product lines that depend on each other. Thus, by contrast to the modeling of software product lines, in which we describe the dependencies between features, the modeling of a multi product line needs to describe the dependencies between different product lines. Even if the dependencies between product lines are also based on the description of feature dependencies, the feature-model complexity is increased, hard to completely understand, and to manage. Thus, the modeling strategy using feature models is insufficient in the scenario of multi product lines.

To overcome the limitations of feature models regarding large-scale and ultra large-scale systems, several concepts were proposed that mainly focus on the usage of multiple feature models with interdependencies to manage the complexity and the configuration of these variable systems [Acher et al. 2013a; Classen et al. 2011; Damiani et al. 2014; Reiser and Weber 2006; Rosenmüller et al. 2011; Rosenmüller et al. 2008]. In detail, feature-model dependencies were analyzed, described, formalized and often used as input for the definition of textual languages that can be used to describe the dependent feature models. For instance, Reiser and Weber introduced the concept of multi-level feature trees to handle similar feature models of the automotive domain using a reference feature model and multiple referring feature models in which relations to the reference model exist (e.g., a mapping of a specific feature) [Reiser and Weber 2006]. This allows the designer to reduce the complexity of the reference model, as not the complete variability is included in this feature model. By contrast, other modeling approaches, such as the concept of Rosenmüller et al., focus on a separation of different functionalities to multiple (parts of) feature models [Rosenmüller et al. 2011; Rosenmüller et al. 2008]. As the concepts of Rosenmüller et al. are the starting point of our work, we use this background section to focus on these techniques.

Rosenmüller et al. use a comparison to OOP to describe the interdependencies of product lines [Rosenmüller et al. 2008]. As one outcome, Rosenmüller et al. compare a product line with an object-oriented class that can also have multiple instances. By aggregation, we can use these instances inside of other product lines and it is possible to utilize uses-relationships between the involved product-line instances. Similar to OOP, we can define a name for a product-line instance to describe relations that only correspond to this instance. In addition, according to Rosenmüller et al. it is also possible to use inheritance for feature models to specialize it. This mechanism is comparable with staged configuration [Czarnecki et al. 2005], in which a feature model is preconfigured to reduce the variability of the system. In sum, the comparison with OOP comes with several advantages that can help to comprehend the relations between the different product lines. For instance, as for OOP several modeling techniques exist, we also can reuse these techniques to ease the comprehension of relations in product lines. Thus, Rosenmüller et al. suggest to use UML class diagrams to describe the dependencies between the involved product lines [Rosenmüller et al. 2008].

Modeling of Multi Product Lines by Example

To illustrate the concepts of Rosenmüller et al., we use an extended version of our running example with product line *BankAccount*. In detail, we plan to reuse the functionality of product line *BankAccount* in a second product line that realizes a bank application. The resulting (multi) product line *BankApplication* should allow us, for instance, to manage different bank accounts, or to transfer money from one bank account to another. In addition, it should be possible to log operations or errors so that subsequent tracing is possible. Before we can start to focus on the running example of the (multi) product line *BankApplication* with all dependencies to other product lines, we need to introduce the features of product line *BankApplication* (without all features of reused product lines) and their functionality. Furthermore, we introduce a product line *List* that can be used for logging. Afterwards, we focus on the dependencies of product line *BankApplication* to the other feature models and possible interdependencies of features.

In Figure 2.11, we depict the feature model of product line *BankApplication*. As mentioned above, the feature *BankApplication* controls the main functionality and manages different bank accounts. Feature *Transaction* provides functionality to transfer money from one bank account to another. In addition, the features *Permission*, *Read*, and *Complete* provide functionality to control the permission of the managed accounts. If the feature *Complete* is included in a product, it is also necessary to include feature *Transaction*. Therefore, the dependency *Complete* $\rightarrow$ *Transaction* is added as cross-tree constraint to the feature model. By contrast, in Figure 2.12, we introduce the features of a product line *List*. In detail, product line *List* provides the features *Synchronize*, *Sort*, *QuickSort*, *MergeSort*, *InsertionSort* for a synchronized and ordered list that can be used for different purposes depending on the specific reuse scenario. In our running example, we could use the functionality of product line *List* to log operations and errors.

The described feature model of the product line *BankApplication* only considers functionality to update bank accounts without a description of the underlying bank-account or logging functionality. To realize the desired bank-account functionality, two options exist. On the one hand, it is possible to design the functionality for a bank account from scratch. This would result in huge effort, in which we have to extend the feature model and need to implement, specify and verify the underlying source code. On the other hand, we can reuse functionality of an existing product line that already provides this functionality. Thus, we can profit from the already existing dependencies of the feature model, and the implementation. The result is a multi product line, in which we have to describe the dependencies between the features of the product line *BankApplication* and the product line *BankAccount*. Afterwards, we can focus on the implementation and the specification of the multi product line *BankApplication*. Similarly, we can treat the extension to introduce the functionality of logging. For this purpose, we also have the choice between the two options of reimplementation or reuse. Here, we decide to reuse the existing functionality from product line *List* that provides the needed functionality (cf. Figure 2.12).

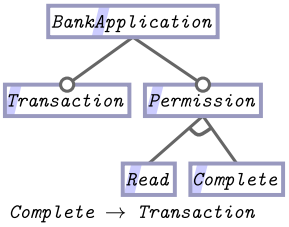


Figure 2.11: Feature model of product line *BankApplication*.

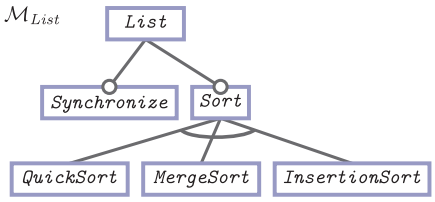


Figure 2.12: Feature model of product line *List*.

Using the concepts of Rosenmüller et al., we can focus on the second option, in which we reuse the functionality of already existing product lines [Rosenmüller et al. 2008]. To describe the dependencies between the involved product lines, we use a kind of class diagrams that we depict in Figure 2.13. In this figure, we represent each product line by a class. In detail, the class representation of product line *BankApplication* reuses the classes *BankAccount* and *SynchronizedList* based on aggregation. The class *SynchronizedList* is a pre-configured product line of product line *List* (cf. Figure 2.12) to ensure that the list can be used to log errors. Therefore, we use inheritance and specialize the subclass *SynchronizedList* so that the feature *Synchronize* is selected. Similarly, other pre-configurations are possible that we can use inside of the involved product lines or completely other product lines. Additionally, we also use aggregation to indicate that the product line *BankApplication* uses an instance of product line *BankAccount*.

To reduce the complexity of our descriptions regarding multi-level interfaces, we only use a small excerpt of the described running example. In detail, in this thesis we focus on the dependencies between the (multi) product line *BankApplication* and the product line *BankAccount* and neglect further features and corresponding dependencies (i.e., constraints) to the product line *List* (cf. Figure 2.13).

Feature-Model Analyses for Multi Product Lines

To ensure correct dependencies and to uncover unintended behavior of the product-line combinations, it is necessary to analyze the complete model of the multi product line. In detail, it is necessary to take a look at all constraints of the multi product line, i.e., constraints because of the tree structure, the intra-model constraints and the inter-model constraints. An intra-model constraint is a constraint that only uses features of the same feature model, without dependencies to features of another feature model. By

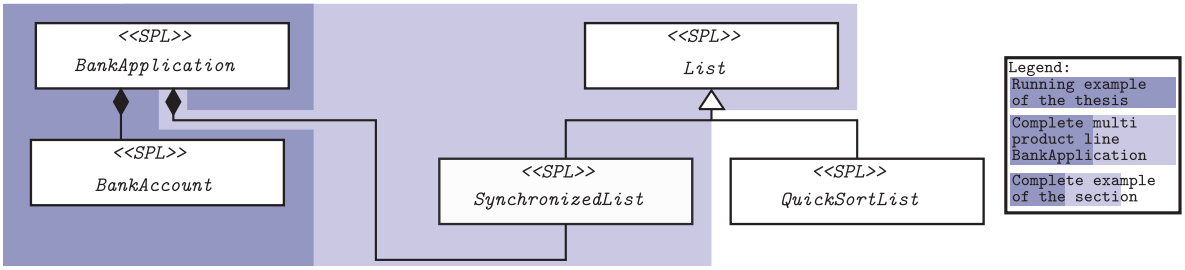


Figure 2.13: Product-line dependencies described by a class diagram.

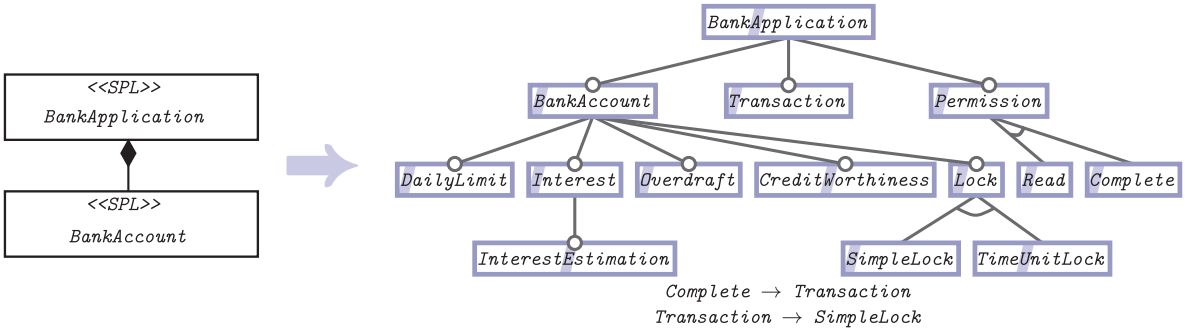


Figure 2.14: Transformation of dependent feature models to one complete feature model (e.g., using aggregation).

contrast, an inter-model constraint is a constraint that uses features of multiple feature models of the multi product line to describe the dependencies. Thus, similar to ordinary product lines, it is important to know whether a combination of features represents at least one valid product.

To consider the correctness of the modeling of a multi product line, the thesis focuses on the improvement of automated analyses that we already presented in the context of ordinary product lines (e.g., the analysis of dead features, cf. Section 2.1.1). In this context, we and others already reused analysis techniques for product lines [Acher et al. 2013a; Schröter et al. 2013b] for multi product lines. Our idea behind can be described as follows. First, we combine all feature-model artifacts of the multi product line to one complete feature model (cf. Figure 2.14, for the analysis process the logical representation of Section 2.1.1 is used). Second, we analyze the complete feature model in an ordinary manner as known from feature models of product lines (cf. paragraph *Feature-Model Analysis* Section 2.1.1). For instance, we use the complete feature model to check whether the feature combinations of the multi product line represent at least one product (cf. Definition 2).

In general, it is possible to reuse all presented analysis techniques of this thesis (cf. paragraph *Feature-Model Analysis* Section 2.1.1) to investigate the correctness of multi product lines. However, in previous work, we have shown that the result of some analysis techniques needs to be considered carefully. For instance, if inheritance is used to pre-configure product lines, features automatically become dead or false-optional

(cf. Definition 4). As a consequence, the corresponding analysis also determines these features as dead and false-optional even if the reason for this anomaly was a desired configuration [Schröter et al. 2013b]. In detail, if we consider the feature dependencies of the product line *QuickSortList* (cf. Figure 2.13), we use inheritance to pre-configure the feature model so that feature *QuickSort* is selected. Using the corresponding analysis, we detect the feature *QuickSort* as false-optional feature and the features *MergeSort* and *InsertionSort* as dead features. To overcome the misleading analysis results, we proposed the analyses *dependent-false-optional feature* and *dependent-dead features* [Schröter et al. 2013b]. These analyses are adapted versions of the corresponding analyses *false-optional feature* and *dead feature* for ordinary feature models and respect desired configuration options of multi product lines. Considering our example of multi product line *QuickSortList*, the analyses result in an empty list of *dependent-false-optional features* and *dependent-dead features*.

2.2.2 Implementation of Multi Product Lines

In Section 2.1.2, we gave an excerpt of multiple techniques to implement single product lines based on annotation-based and composition-based approaches. As the implementation techniques (e.g., preprocessors such as Antenna [Pleumann et al. 2011], FOP [Batory et al. 2004; Prehofer 1997], AOP [Kiczales et al. 1997], DOP [Damiani et al. 2014; Schaefer et al. 2010]) and programming languages (e.g., C++, Java) for product lines are diverse, the combination of product lines to form a multi product line can become a complex task. In detail, multiple programming techniques and languages can be used to implement the single product-line systems. However, the complexity of the implementation task of a multi product line not only depends on the implementation techniques and languages but also on the kind of product-line dependencies, such as inheritance and aggregation. In this section, we present some details about this problem and the resulting variety of solutions. In this context, we isolate the relevant techniques that are focused in this thesis.

Multi Product Lines Based on Multiple Programming Languages

As described above, it is a hard task to implement a multi product line as the dependent product lines can be implemented in different programming languages and with different implementation techniques. Here, we take a look at the problem when using different programming languages. Considering ordinary programs without variability, the problem of combining different programming languages is well known and comes with several challenges and pitfalls, such as pitfalls regarding refactorings [Schink et al. 2016]. In general, the combination of languages can be cumbersome (i.e., integration of C++ in Java). Thus, it is obvious that a combination of product lines based on different programming languages is in general a more complex task as product lines based on the same programming language. Consequently, the combination of multiple programming languages with corresponding interactions is a separate research area that we do not focus in this thesis. In detail, we present our concepts regarding the implementation of multi product lines using dependent product lines that are based on the same programming language (i.e., Java).

Multi Product Lines Based on Multiple Implementation Techniques

Even if we use the same programming language for product lines, it is not ensured that the product lines can be easily combined to form a multi product line. Because of the variety of implementation techniques (i.e., annotation-based approaches and composition-based approaches) and possible interactions of the involved product lines (i.e., dependencies of inheritance or aggregation), diverse combinations of implementations are possible. However, not all combinations are feasible. In the following, we consider examples for inheritance and aggregation.

As an example for inheritance, we consider product line *List* (cf. Figure 2.12) that we plan to extend by a new sort algorithm. On the modeling level, this is an easy task, in which we create a new product line *ExtendedList* that uses inheritance to inherit all modeling characteristics of the already existing product line *List*. Afterwards, we add a new feature to product line *ExtendedList* that represents the new sort algorithm. By contrast, on the implementation level, the realization of the new feature depends on the implementation technique and the code structure of the underlying product line with its corresponding characteristic. For instance, if product line *List* is based on an annotation-based approach, it also depends on the source-code location of the existing algorithm whether it is difficult to implement this alternative sort algorithm in a separate product line. Furthermore, if the product line *List* is based on a composition-based approach, such as AOP [Kiczales et al. 1997], FOP [Batory et al. 2004; Prehofer 1997], or DOP [Damiani et al. 2014; Schaefer et al. 2010] several solutions exist to create this new sort feature in the separate product line *ExtendedList*. For instance, if AOP is used to implement product line *ExtendedList*, we can easily define a pointcut to implement a new behavior of the sort algorithm of the already existing product line *List*. Since the pointcut can be used to reference a particular operation in the source code, it does not matter if the sorting was separated in a dedicated method or whether it was included in other source-code artifacts. Similar extensions are possible if FOP or DOP is used. However, for FOP it is necessary that the sorting was separated in a respective method. Based on this, we can define a new sort algorithm that overrides the sort algorithm of the original product line *List*. If such a method does not exist, it is not possible to override the behavior without a refactoring inside of the original product line *List*.

By contrast to inheritance, the mechanism of aggregation enables a more flexible combination of implementation mechanisms. As illustration, we use our running example of multi product line *BankApplication* with product line *BankAccount*. On the modeling level, the multi product line *BankApplication* instantiates the product line *BankAccount* to reuse the corresponding functionality of the product line *BankAccount* (cf. left side of Figure 2.14). On the implementation level, we can consider the product line *BankAccount* as a library that provides reusable functional artifacts that we can use in the multi product line *BankApplication*. This allows us to treat the product line as an ordinary library of the specific programming language that we can use in an ordinary manner. However, the developer needs to be aware of the variability of the library as not all API members, such as classes, methods, and fields are available in all

feature-combinations of the reused product line *BankAccount* (for more details, we refer the reader to Chapter 3 and Chapter 5). In sum, the mechanism of aggregation that uses libraries on the implementation level results in an advantage for the developer as the implementation mechanism behind the reused product line can be hidden. Thus, it does not matter for the developer of the multi product line *BankApplication* whether the product line *BankAccount* is based on an annotation-based or composition-based approach.

The previous example of aggregation is based on a single instantiation of the product line *BankAccount* inside of the multi product line *BankApplication*. By contrast, it is also possible that more than one instance of a specific product line with different configurations is used inside of another product line. This complicates the reuse scenario on the implementation level as it is not possible to include a library in different variants at the same time. Rosenmüller et al. consider this problem and presents several approaches to solve this problem on the implementation level [Rosenmüller et al. 2010]. One solution is based on a namespace concept, in which each product-line instance can be addressed by a unique qualified name. Thus, if more than one product-line instance is reused in another product line, the instances need to be refactored so that the namespace is unique again.

Focused Language and Implementation Technique

In this thesis, we focus on multi product lines that are based on aggregation and, thus, we consider the reuse of a product-line variant as library inside of another product line. Even if it is possible to reuse a product line as library with variable artifacts inside of other product-line implementations, it is still a challenging task to implement a multi product line without compile-time errors. Since, it exists a direct dependency between the involved product lines so that changes in the underlying product line directly influence the source code of product lines that depend on this implementation.

2.3 Summary

In this chapter, we presented background information that are necessary to follow our explanations regarding the concept of multi-level interfaces. In a first part, we described the procedure on how to develop a software product line. Therefore, we considered the concept of feature models that can be used to describe the dependencies between the features of a product line. Afterwards, we gave an excerpt on implementation mechanisms for a software product line and illustrated how to verify the system based on design by contract. In a second part, we gave an overview of concepts to develop a multi product line. In this context, we mainly focused on the concepts of Rosenmüller et al. as these concepts represent the base of our investigations. In detail, we presented how to model and analyze the dependencies between the involved product lines of the multi product line using ordinary feature models. Afterwards, we gave an excerpt how to implement multi product lines and how the implementation depends on the underlying implementation concepts of the involved product lines.

For the purpose of illustration, we used our running example of the product line *BankAccount* that we partly extended according to the specific exemplification. To illustrate the concept of multi product lines, we extended the running example using (multi) product line *BankApplication* that we will also use as an example in the remaining thesis. In detail, in the next chapter, we use the background information and the running example to present an overview of our multi-level interfaces. Afterwards, we use our running example to consider each level of our multi-level interfaces in detail. In addition to the description of the main concepts on which this thesis is based on, we describe further techniques for product lines and multi product lines in our related work chapter.

3. An Overview - The Concept of Multi-Level Interfaces

> The chapter presents an overview of our concept of multi-level interfaces. It shares material with papers *Towards Modular Analysis of Multi Product Lines* [Schröter et al. 2013a] and *Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines* [Schröter 2014] that present the initial idea of this concept. In addition, it shares material with the paper *Variability Hiding in Contracts for Dependent Software Product Lines* [Thüm et al. 2016] to describe possible benefits when using each interface.

In this thesis, we propose a general approach to ease the development and analyses of multi product lines using the concept of interfaces. In detail, we propose interfaces on multiple levels to achieve benefits during the development, analyses, and evolution of multi product lines. Therefore, we refine our initial ideas (cf. descriptions of [Schröter et al. 2013a]) so that we ease the development of a multi product line. First, we give an overview of our main concept using our running example. Second, we introduce our hypotheses regarding each interface level that we use in the subsequent chapters to investigate our interface levels.

3.1 An Introduction of Multi-Level Interfaces

In the previous chapter, we used the example of the (multi) product line *BankApplication* and product line *BankAccount* for the purpose of illustration. In *BankApplication*, all dependent and used product lines are known from the point of view of the product line *BankApplication*. This means, that we as developer of product line *BankApplication* have dependencies to all artifacts (modeling, implementation and specification) of all dependent product lines even if not all these parts are of our interest. This can complicate the comprehension and development of the multi product line and hinders an

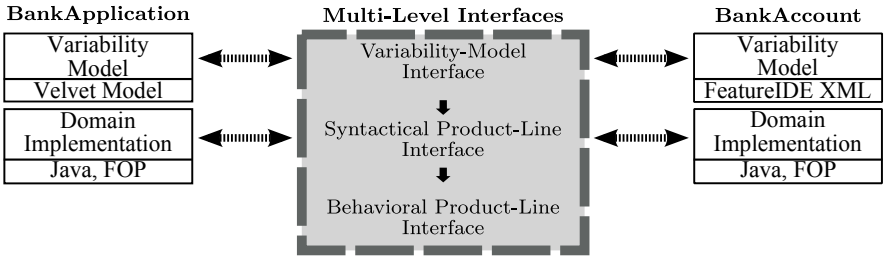


Figure 3.1: Interfaces for the reuse of SPLs in MPLs (adapted version of [Schröter et al. 2013a]).

efficient evolution as we have to reanalyze the multi product line even if the changed feature is not of our interest.

To overcome the drawback that comes with direct dependencies between dependent product lines, we introduce the concept of multi-level interfaces. In detail, we define the (1) *variability-model interface*, (2) *syntactical (product-line) interface*, and (3) *behavioral (product-line) interface*. Assuming an already existing product line (e.g., product line *BankAccount*) with a variability model, a feature-based implementation and specification that we want to reuse in another product line (e.g., product line *BankApplication*), we can describe the interfaces as follows. The variability-model interface is a reduced variability model that only consists of the features that we need in our reuse scenario but it is still conform to the underlying variability model of the already existing product line. The syntactical interface depends on the upper-level interface (i.e., the variability model) and, thus, it only consists of implementation artifacts (e.g., API members) of the underlying product line that are available when the features of interest are used. In addition, the behavioral product-line interface depends on all upper-level interfaces and presents specifications regarding the behavior of available implementation artifacts of the upper-level interface (i.e., the syntactical interface). In Figure 3.1, we give an overview of our concept of multi-level interfaces. Using these interfaces, we aim to focus on artifacts of a particular level that have dependencies to other artifacts of involved product lines of the multi product line. All other artifacts should be hidden by the interface of the specific level.

In the subsequent sections, we give a brief overview of the main ideas regarding our concept of multi-level interfaces. By contrast, we present details on each interface in the following chapters, and give details on their functionality, advantages, and generation strategies. In detail, Chapter 4 focuses on the feature-model interfaces, Chapter 5 on feature-context interfaces, and Chapter 6 on behavioral product-line interfaces.

3.2 The Feature-Model Interface as Variability-Model Interface

The feature-model interface is a realization of the originally introduced conceptual idea of *variability-model interfaces* [Schröter et al. 2013a]. A variability-model interface is

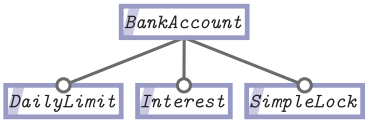


Figure 3.2: Feature-model interface for the product line *BankAccount* as used in product line *BankApplication*.

itself a variability model and it is used to define an agreement between two involved product lines of a multi product line [Schröter et al. 2013a]. This means, a developer of a specific product line who wants to use another product line needs to define the artifacts of interest that are needed in the reuse scenario. For feature-model interfaces, the artifacts of interest are the features that should be reused [Schröter et al. 2016]. Therefore, a developer needs to define all features that are needed in a specific reuse scenario inside of a multi product line.

Running Example

For instance, let us consider the reuse scenario of multi product line *BankApplication*. In this scenario, the original feature model *BankAccount* consists of nine different features (cf. Figure 2.1) but besides the root feature, only the features *DailyLimit*, *Interest*, and *SimpleLock* are of our interest for a reuse in a multi product line with *BankApplication*. Thus, we plan to create a feature-model interface as an agreement so that we have the guarantee that these features exist with the given dependencies. At the same time, we consider the other features, such as feature *Overdraft* and feature *InterestEstimation*, as not relevant and, thus, it doesn't matter whether these features are included in the final product. In Figure 3.2, we present the feature-model interface that is used as an agreement between both product lines, in which only our relevant features exist. Using the feature-model interface instead of the original feature model *BankApplication*, we are able to reduce the feature-model complexity. As a result, the complexity of combinations with other feature models like *BankApplication* is also reduced. Therefore, we expect a significant performance improvement of feature-model analyses regarding combined feature models if we use dependencies to the feature-model interface instead of direct dependencies to the original feature model of the product line *BankAccount*. In particular, we also expect that it is not necessary to reanalyze the multi product line *BankApplication* and product line *BankAccount* for all evolutional changes of product line *BankAccount*.

3.3 The Variable Interface as Syntactical Interface

The variable interface is based on our ideas according to the proposed *syntactical (product-line) interface* [Schröter et al. 2013a]. The syntactical interface represents an API with variability information [Schröter et al. 2013a]. In detail, the interface consists of signatures, such as classes, methods, and fields that are accessible by different configurations of a specific product line. As all interfaces of our concept of multi product

$$
\begin{aligned}
&\mathcal{VI}_{BankAccount} = \{ \\
&\quad (class\ Account, \{B, D, I, O, W, E, S\}), \\
&\qquad (Account.calculateInterest() : int, \{I\}), \\
&\qquad (Account.credit(int) : boolean, \{W\}), \\
&\qquad (Account.estimatedInterest(int) : int, \{E\}), \\
&\qquad (Account.getOverdraftLimit() : int, \{B, O\}), \\
&\qquad (Account.isLocked() : boolean, \{S\}), \\
&\qquad (Account.lock() : void, \{S\}), \\
&\qquad (Account.unlock() : void, \{S\}), \\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D\}), \\
&\qquad (Account.update(int) : boolean, \{B, D\}), \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 3.3: Variable interface of product line *BankAccount* (excerpt of class `Account` without fields).

lines are based on each other, we only consider configurations that are defined in the modeling agreement represented by the feature-model interface. As a result, the syntactical interface is also an agreement between the involved product lines, in which the API members that we can reuse are well defined. However, we present the *variable interface* as one specific implementation of the syntactical interface [Schröter et al. 2014]. The variable interface was designed for product lines based on the programming language Java with Feature-Oriented Programming (FOP) as implementation paradigm. It provides all necessary information as proposed by our concept of syntactical interfaces. Based on this, we are also able to provide feature-context interfaces that provide a special and non-variable view to the variable interface according to a specific implementation context (i.e., in the context of FOP, a specific feature that we currently implement). As a result, using the feature-context interface, it is easier for the developer to comprehend the source code and, thus, to implement the product line in a more efficient way.

Running Example

For the purpose of illustration, we take a look at our running example of the multi product line *BankApplication* with product line *BankAccount*. In this scenario, the product line *BankApplication* needs to call functionality of the product line *BankAccount*. Therefore, the developer needs to know which API members can be called without compile-time errors. For this purpose the developer also needs to know the dependencies inside of the product line *BankAccount* and all dependencies to the product line *BankApplication*. To get support for this problem, we provide the variable interface $\mathcal{VI}$ and the feature-context interface that was designed for the programming language Java with FOP. First of all, we generate the variable interface, in which all information

$$
\begin{aligned}
&\mathfrak{VI}_{BankAccount} = \{ \\
&\quad (class\ Account, \{B, D, I, S\}), \\
&\qquad (Account.calculateInterest() : int, \{I\}), \\
&\qquad (Account.getOverdraftLimit() : int, \{B\}), \\
&\qquad (Account.isLocked() : boolean, \{S\}), \\
&\qquad (Account.lock() : void, \{S\}), \\
&\qquad (Account.unlock() : void, \{S\}), \\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D\}), \\
&\qquad (Account.update(int) : boolean, \{B, D\}), \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 3.4: Filtered variable interface of product line *BankAccount* (excerpt of class `Account` without fields).

of the API members and their accessibility are included. The initial variable interface includes all variability information with all features that we can filter according to the features of interest given by the feature-model interface. Afterwards, we can use this variable interface to generate a feature-context interface to tailor the set of accessible API members to the current implementation context (i.e., the feature).

In Figure 3.3, we present the initial variable interface of product line *BankAccount* that was created using the corresponding code base of product line *BankAccount*. The variable interface itself is an array with the information about all product-line API members with additional information. In detail, for each API member, we get the information about features, in which the specific API member is defined (cf. highlighted characters of features given in Figure 2.1). For instance, the class `Account` is defined in all concrete features of the product line *BankAccount*. By contrast, the class member `Account.update` is only defined in the features *BankAccount* (*B*), and *DailyLimit* (*D*). According to the features of interest given in the feature-model interface (cf. Figure 3.2), we can now filter the variable interface so that only API members of our interest are represented. Whereas multiple filtering techniques are possible, we present the result of our simple filtering technique (cf. Figure 3.4). As result of the simple filtering, we only present members that are defined in the features of interest. All other members, like member `Account.credit` of feature *CreditWorthiness*, were removed. However, other filtering techniques are possible and we will discuss these approaches in the corresponding chapter (see Section 5.3).

The feature-context interface is based on the variable interface and has further advantages according to the product-line development. In detail, the feature-context interface is able to support the developer during the maintenance and development process of a product line giving a list of accessible API members that can be used in a specific development task. By contrast to the variable interface, the feature-context interface

$$
\begin{aligned}
&\mathcal{FCI}_{Transaction} = \{ \\
&\quad class\ Account, \\
&\qquad Account.getOverdraftLimit : int, \\
&\qquad Account.isLocked() : boolean, \\
&\qquad Account.lock() : void, \\
&\qquad Account.unlock() : void, \\
&\qquad Account.undoUpdate(int) : boolean, \\
&\qquad Account.update(int) : boolean, \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 3.5: Feature-context interface of product line *BankAccount* for feature *Transaction* of product line *BankApplication* (excerpt of class `Account` without fields based on the simple variable interface $\mathcal{VI}_{BankAccount}$).

presents a non-variable view to the variable interface. This means, the interface does not contain variability information, it only consists of a list of accessible API members. For instance, assuming that a developer wants to reuse API members of the product line *BankAccount* inside of code artifacts of the feature *Transaction* of product line *BankApplication*. For this purpose, we can generate a feature-context interface in which all accessible API members of the product line *BankAccount* are represented. In Figure 3.5, we present the resulting feature-context interface for the feature *Transaction* according to elements of product line *BankAccount*. As we can see, the feature-context interface is only a list of members that can be safely called from code artifacts of feature *Transaction* in product line *BankApplication*. This means, it presents only API members that are accessible in all products in which also the feature *Transaction* is included. As result, the API member `Account.calculateInterest` is not included because this member is only accessible if feature *Transaction* forces the selection of feature *Interest*.

3.4 The Behavioral Product-Line Interface

As a proof of concept to investigate whether it is possible to extend our multi-level interface to further levels as modeling and implementation, we also introduce the *behavioral (product-line) interface*. The behavioral interface is the third interface of our multi-level interfaces (cf. Figure 3.1) and provides specifications about the behavior of methods behind this interface. Like the variable interface, the behavioral product-line interface also depends on the interfaces from the upper levels. Thus, the behavioral product-line interface is based on the variable interface as well as the feature-model interface and specifies the behavior of each available method [Schröter et al. 2013a]. Afterwards, we can use these interface specifications to verify the behavior of the product line that is supposed to use this interface. Using this behavioral product-line interface, we assume that it is possible to enable a modular verification so that it is not necessary to consider the specification of the whole multi product line.

```
1 class Account {
2   //further source code of the metaproduct
3
4   //@ ensures !FM.overdraft ==> \result == 0;
5   //@ ensures FM.overdraft ==> \result == -5000;
6   int /*@ pure @*/ getOverdraftLimit(){
7     if (!FM.overdraft) return 0;
8     return -5000;
9   }
10 }
```

(a) Method `getOverdraftLimit` in the metaproduct of product line *BankAccount*.

```
11 class Account {
12   //further source code of the metaproduct
13
14   //@ ensures (\result == -5000) || (\result == 0);
15   int /*@ pure @*/ getOverdraftLimit(){
16     if (!FM.overdraft) return 0;
17     return -5000;
18   }
19 }
```

(b) Method `getOverdraftLimit` with a simplified postcondition after the application of strategy *hidden configuration*.

Figure 3.6: Method `getOverdraftLimit` of class `Account` represented in the metaproduct before and after the application of the strategy hidden configuration.

For the specification of the behavioral product-line interface, we only consider API members of the variable interface and the specification should only include dependencies to features that are represented in the feature-model interface. In theory, the concept of the behavioral product-line interface is a general concept that in practice depends on the implementation language and the used product-line paradigm. For our proof of concept, we use the technique of design by contract [Meyer 1988, 1992] for the programming language Java with FOP [Thüm et al. 2012] (for more details, we refer the reader to Chapter 6).

Running Example

We use our running example of the multi product line *BankApplication* with product line *BankAccount* to illustrate the idea of behavioral product-line interfaces. In detail, we take a look at the method `getOverdraftLimit` as it is still a method of the filtered variable interface and it is part of our features of interest (cf. Figure 3.2 and Figure 3.4). The method itself exists in the original product line *BankAccount* in two different features, in feature *BankAccount* with return value `0` and in feature *Overdraft* with return value `-5 000`. However, the feature *Overdraft* is not part of the features of interest and, thus, not included in the filtered variable interface (cf. Figure 3.4). As a result, we also need to remove the feature from the contracts of the metaproduct (for more details on metaproducts, we refer the reader to Section 2.1.3) to create a behavioral product-line interface. Therefore, the task is to get a suitable representation of the original

contracts according to method `getOverdraftLimit` so that the behavioral product-line interface can be used for correctness proofs of product line *BankApplication* instead of the original product line *BankAccount*.

It is possible to generate an initial behavioral product-line interface based on the existing code and contracts of the product line *BankAccount*. Therefore, we have to consider all contracts according to the method `getOverdraftLimit` using the metaproduct of the product line *BankAccount*. In Figure 3.6 (a), we present the contract of the metaproduct for the method `getOverdraftLimit`. In detail, if we take a look at the ensures clauses, we can see that the method result depends on the feature *Overdraft* so that `0` will be ensured without and `-5 000` with the feature *Overdraft*. However, the feature *Overdraft* is not of our interest and, thus, not a member of the feature-model interface. Therefore, we have to remove feature *Overdraft* from the specification. In Chapter 6, we present several strategies to create a behavioral product-line interface in which the result is a contract without feature *Overdraft*. Here, we use the strategy *hidden configuration* to adapt the specification. In Figure 3.6 (b), we present the result of the application of this strategy, in which the new contract specifies that the method returns either `0` or `-5 000`. Afterwards, we can use this contract to verify methods of product line *BankApplication* that call the method `getOverdraftLimit`.

3.5 Analyzing the Benefits of Multi-Level Interfaces

Based on the previous introduction of our multi-level interfaces, we give an overview of the potential benefits. Therefore, we summarize our goals that we aim to achieve with our concept of multi-level interfaces and present potential benefits for each interface. Afterwards, we give details on benefits in the case of product-line evolution. Finally, we introduce hypotheses for each interface that we want to investigate in the remaining thesis.

In general, using multi-level interfaces, we aim to avoid direct dependencies between product lines. As result, we want to ease the analyses and the evolution of the multi product line. For this purpose, we take a look into the detailed goals for each interface level.

Considering the variability-model interface, the goal for the feature-model analysis is to focus on features that we are interested in. All other features can be neglected. Thus, the product line behind the variability-model interface can be analyzed solely if the developer is interested in the results, but only the variability-model interface is needed to analyze the variability model that reuses functionality of the hidden product line. For instance, in our multi product line *BankApplication*, we can solely analyze the feature model *BankAccount*. Using the concept of the feature-model interface, we can focus on the features of interest and we can analyze the feature-model composition consisting of product line *BankApplication* and the feature-model interface (cf. Figure 3.2) without the knowledge of the complete feature model of product line *BankAccount*.

The goal of the syntactical interface is to modularize the product-line implementation due to a presentation of specific implementation artifacts (e.g., methods or fields) that

can be reused in the (multi) product line. For this purpose, the variable interface summarizes all existing implementation artifacts that can be used in a specific product line when the features of the upper-level variability-model interface are selected. Furthermore, the variable interface additionally presents variability information to reason about a correct usage of all these implementation artifacts. Based on this, we also propose to use a non-variable view to the variability-model interface that only presents all safely accessible implementation artifacts. As a result, using the variable interface and the non-variable view to its artifacts, we aim to ease the product-line development through user support. For instance, for the concept of FOP, we propose the corresponding variable interface and the feature-context interface as non-variable view. Based on this, the developer of product line *BankApplication* gets an overview of all safely accessible members that can be reused in the current developed feature module. In detail, the developer of the feature module *Transaction* gets an overview of safely accessible members for this feature module, whereas the method `update` is part of it (cf. Figure 3.5). As a result, the usage of method `update` does not lead to a compile-time error.

The goal of the behavioral product-line interface is to ease the verification effort of multi product lines due to modularization. Considering the concept of design by contract, the behavioral product-line interface only focuses on features that are available in the variability-model interface. As a result, a product line that wants to reuse the corresponding features can use the behavioral product-line interface for the own verification in which all dependencies to other features are hidden. For instance, we can solely verify the product line *BankAccount*. In an optimal case, we can use product line *BankAccount* to generate a behavioral product-line interface tailored to the features that are available in the feature-model interface so that it is not necessary to reverify the behavioral product-line interface. However, even if it is necessary to reverify the behavioral product-line interface, we can use the interface to verify the product line *BankApplication* without the knowledge of the complete contracts that exist in the product line *BankAccount*. Nevertheless, it is an open question whether a behavioral product-line interface can reduce the verification effort of a multi product line.

By contrast to the benefits through the initial development of multi product lines, we also assume potential benefits for the evolution of multi product lines when using multi-level interfaces. We already presented potential benefits of the behavioral product-line interface regarding the product-line evolution [Thüm et al. 2016] that we now extend to all interfaces of our multi-level interfaces. We use our running example to illustrate the benefits for each interface. Four different scenarios exist for the evolution of a multi product line that we want to illustrate in detail. In this context we assume that we are interested in the analysis results of the product line *BankAccount* and the multi product line *BankApplication*.

Case 1. Changes on a specific development level of product line *BankAccount* do *not* require to change the corresponding interface:

- If changes occur in the feature model, we have to reanalyze the feature model *BankAccount*;

- Regarding changes on the syntactical level, the variable interface of product line *BankAccount* needs to be updated;
- Changes on the product-line behavior force the reverification of product line *BankAccount*.

Case 2. Changes on a specific development level of product line *BankAccount* require to change the corresponding interface:

- Changes regarding the feature model force to reanalyze the feature model *BankAccount*. In addition, it is necessary to renew the feature-model interface and the feature-model composition with product line *BankApplication* and to reanalyze it;
- If changes on the syntactical level occur, we need to update the variable interface of product line *BankAccount*, to apply the filtering techniques to the features of interest, and to analyze the impact on product line *BankApplication*;
- For changes on the product-line behavior, product line *BankAccount* needs to be reverified and the behavioral product-line interface is generated to reverify product line *BankApplication*.

Case 3. Changes on a specific development level of product line *BankApplication* do *not* require to change the corresponding interface:

- If the changes occur on the feature model, we have to reanalyze the feature-model composition of product line *BankApplication* based on the feature-model interface;
- Changes on the syntactical level force to update the variable interface of product line *BankApplication*;
- For changes on the product-line behavior, it is necessary to reverify product line *BankApplication* using the behavioral product-line interface.

Case 4. Changes on a specific development level of product line *BankApplication* require to change the corresponding interface:

- If the changes are required on the modeling level, we have to regenerate the feature-model interface, update and reanalyze the feature-model composition;
- Required changes on the syntactical level force us to refilter the variable interface of product line *BankAccount* and to update the variable interface of product line *BankApplication*;
- For required changes on the product-line behavior, we have to generate the behavioral product-line interface to reverify product line *BankApplication*.

In summary, only Case 2 forces us to renew all artifacts and, thus, to reapply the corresponding modularization concept. In all other evolution scenarios of the multi product line it is possible to achieve benefits using our interface concept compared to concepts that use direct product-line dependencies.

Introducing Hypotheses to Investigate the Potential Benefits

As illustrated above, we assume a huge amount of potential benefits when using the concept of multi-level interfaces. However, to concretize our investigations described in this thesis and to answer our global research questions, we formulate one hypothesis for each interface of our concept. According to these interfaces, we organize the evaluation and corresponding discussions of each interface that we present in the following chapters.

Hypothesis 1. *Potential of Variability-Model Interfaces* according to *Hypothesis H1* of [Schröter 2014]**:**

The variability-model interface enables a performance improvement of automated analyses on variability models in evolving Multi Product Lines (MPLs).

Hypothesis 2. *Potential of Syntactical Product-Line Interfaces* according to *Hypothesis H2* of [Schröter 2014]**:**

The syntactical product-line interface helps to detect reusable code artifacts and reduces the development time compared to state-of-the-art techniques.

Hypothesis 3. *Potential of Behavioral Product-Line Interfaces* according to *Hypothesis H3* of [Schröter 2014]**:**

The behavioral product-line interface based on design by contract enables a time-efficient modular analysis to detect violations in MPLs using verification techniques.

3.6 Summary

In this chapter, we presented the conceptual idea and gave an overview of our concept of multi-level interfaces. We introduced the idea of each interface and gave details on how the interfaces are interconnected, and how they can be used. Afterwards, we gave an overview of goals for each interface and present potential benefits when using our multi-level interfaces in cases of evolution. To enable a clear investigation and discussion of our concept, we formalize one hypothesis for each interface based on the summarized potential benefits.

In the following three chapters, we introduce the conceptual details for each interface of our multi-level interfaces. In detail, in Chapter 4, we introduce details of the variability-model interface. Using the feature-model interface as realization of the variability-model interface, we show relations between the results of feature-model analyses for multi product lines with and without feature-model interfaces. Based on this theoretical reflection, we start an evaluation of the feature-model interface and investigate our Hypothesis 1. Similarly, Chapter 5 investigates the syntactical interface. In detail, we use the variable interface as representation of the syntactical interface and use this proposed concept with the concept of feature-context interfaces to investigate Hypothesis 2. In Chapter 6, we use the behavioral product-line interface as proof of concept, in which we plan to investigate whether our concept of multi-level interface can also be applied to advanced concepts. Thus, we present different ideas how a behavioral product-line interface can look like. According to Hypothesis 3, we investigate the potential for each idea.

4. The Variability-Model Interface - The First Level of Multi-Level Interfaces

The chapter about *variability-model interfaces* shares material with two papers. First, it shares material with the overview paper *Towards Modular Analysis of Multi Product Lines* [Schröter et al. 2013a], which describes the initial ideas of an interface for variability models. Second, we refined our initial ideas of a variability-model interface in the paper *Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems* [Schröter et al. 2016]. In this paper and in the corresponding technical report, we present proofs of analysis-result relations and present resulting benefits based on a real-world evaluation.

In this chapter, we take a look at the first interface of our multi-level interface concept (cf. Chapter 3) that we use to avoid direct dependencies on the modeling level of multi product lines. In detail, we consider the general concept of our variability-model interface and the facility to ease the analysis of dependent feature models, especially in cases of evolution. To present more conceptual details, we exemplify the ideas by introducing a feature-model interface as a special kind of a variability-model interface for feature models. However, the feature-model interface (resp. the variability-model interface) is the first and the most important interface of our multi-level interface concept on which all other interfaces depend directly. This means, all other interfaces take the information given by the feature-model interface for the purpose of creation or further usage of the specific interface. Therefore, we take special attention to this interface and we pick up this topic in all of the remaining chapters and show how these interfaces are connected.

The chapter is structured as follows. First, as the feature-model composition is essential to enforce the analysis advantages of our concept regarding feature-model interfaces, we respectively present a formalization of feature-model compositions. In this context, we also formalize the feature-model interface and present properties about the interaction of feature-model composition and feature-model interfaces. Second, considering selected analyses (cf. Definition 2–6), we look at the detailed advantages of feature-model interfaces and prove analysis-result relations between feature-model compositions with and without feature-model interfaces. Third, we investigate the practical relevance of our theoretical investigation by using a real-world case study with two application scenarios. Finally, we discuss our evaluation results and summarize our ideas.

4.1 Feature-Model Composition Meets Interfaces

In the introduction and background chapter, we have seen that the state-of-the-art procedure to analyze dependent feature models results in huge effort and recomputations especially in the case of evolution. However, before we can start to discuss and prove how we can benefit using feature-model interfaces with feature-model composition, we need to formalize the concepts. We start with a formalization of feature-model composition followed by the formalization of the feature-model interface. Afterwards, we give a detailed look at the properties resulting from a combination of both concepts that are essential for our proofs presented in the next section.

4.1.1 Concept of Feature-Model Composition

Feature-model composition enables to combine multiple feature models in different manners [Acher et al. 2010, 2013b; Bošković et al. 2011; Classen et al. 2011; Rosenmüller et al. 2008], such as inclusion or aggregation. The result is a new feature model in which the artifacts of all input feature models are combined according to the used composition concept. Afterwards, we can use the combined feature model to analyze whether all dependencies correspond to our intention (see also Section 2.2.1, paragraph *Feature-Model Analyses for Multi Product Lines*). In this thesis, we use the concept of *aggregation* for feature-model composition as it was introduced for the variability-modeling language VELVET [Rosenmüller et al. 2011; Rosenmüller et al. 2008; Schröter et al. 2013b]. As a reminder, the aggregation mechanism of VELVET allows a developer to instantiate a feature model inside of another feature model and to add additional inter-model dependencies subsequently. However, the authors do not present a formal description of this composition mechanism. As a formalization is needed for our proofs in the remaining thesis, we formalize this feature-model composition.

To illustrate the mechanism, we use our running example of the product line *BankApplication* (cf. Figure 2.11). In detail, we combine the feature model *BankApplication* with the feature model *BankAccount* (cf. Figure 2.1) using the root feature of product line *BankApplication* as starting point for this aggregation. In addition, we also plan to include the inter-model constraint $Transaction \rightarrow SimpleLock$ as dependency between

both models (cf. Figure 2.14). In our formalization, we use the additional feature model $\mathcal{M}_C$ to describe these dependencies.

According to the aggregation mechanism used by the modeling language VELVET [Rosenmüller et al. 2011] (see Section 2.2.1 for more details), we formalize feature-model composition as follows:

Definition 8. Feature-model composition according to *Definition 3* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, and $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$ be feature models with $\mathcal{F}_C \subseteq \mathcal{F}_x \cup \mathcal{F}_y$. We define the function composition $\circ$ *using $\mathcal{M}_x, \mathcal{M}_y$, and $\mathcal{M}_C$ with infix notation $\circ_{\mathcal{M}_C}$ based on the join function $\bullet$ and function $\mathcal{R}$ to achieve the composed feature model $\mathcal{M}_{x/y}$:*

$$\mathcal{M}_{x/y} = \circ(\mathcal{M}_x, \mathcal{M}_y, \mathcal{M}_C) = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y = (\mathcal{M}_x \bullet \mathcal{R}(\mathcal{M}_y)) \bullet \mathcal{M}_C \tag{8.1}$$

$$\mathcal{R}(\mathcal{M}_y) = \mathcal{R}((\mathcal{F}_y, \mathcal{P}_y)) = (\mathcal{F}_y, \mathcal{P}_y \cup \{\emptyset\}) \tag{8.2}$$

$$\mathcal{M}_x \bullet \mathcal{M}_y = (\mathcal{F}_x, \mathcal{P}_x) \bullet (\mathcal{F}_y, \mathcal{P}_y) = (\mathcal{F}_x \cup \mathcal{F}_y, \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \tag{8.3}$$

The composition function $\circ$ computes a combined feature model using the function R (remove core property) and the function $\bullet$ (join). In the following, we take a look at each function:

R The function R is used to remove the core property of a feature model. In detail, the function takes a feature model as input and creates a new feature model as output. The resulting feature model consists of the same set of features and the set of products additionally includes the empty product. Thus, the empty product is also a valid product of this new feature model.

- The join function represents the core functionality of our composition function $\circ$ and takes two feature models as input to produce a combined feature model as output. Therefore, the function $\bullet$ acts as a join function known from relational algebra [Codd 1970] and the result depends on comparative criteria. In our case, the result depends on the comparison of $p \cap \mathcal{F}_y = q \cap \mathcal{F}_x$. Thus, we can group the behavior of the function accordingly:

  $\bowtie$ In this scenario, the set of features of the input feature models are *not* disjoint. As a result, the function $\circ$ combines all products that fulfill the comparative criteria.

  $\times$ In this case, the set of features of the input feature models are disjoint. As a result, our comparative criteria also evaluates to true since both sides of

the formula are equal to the empty set ($p \cap \mathcal{F}_y = q \cap \mathcal{F}_x = \emptyset$). As a result, the function $\circ$ represents a cross product that is also known from relational algebra and, thus, it combines all products from the first with all products from the second feature model.

- The composition function uses three feature models as input (cf. Equation 8.1). The feature models $\mathcal{M}_x$ and $\mathcal{M}_y$ are the feature models that we want to combine and a third feature model $\mathcal{M}_c$ is used to describe additional constraints that are necessary for the combination of the other input feature models ($\mathcal{M}_x$, $\mathcal{M}_y$). In detail, feature model $\mathcal{M}_c$ consists of a parent-child constraint, which describes the place for the inclusion of $\mathcal{M}_y$ into $\mathcal{M}_x$, and user-defined cross-tree constraints for the feature-model combination. Using this input, the function $\circ$ creates a new combination of the feature models $\mathcal{M}_x$ and $\mathcal{M}_y$ with additional (cross-tree) constraints given by $\mathcal{M}_c$ (i.e., $\mathcal{M}_c$ consists of all inter-model constraints). However, to instantiate feature model $\mathcal{M}_y$ into feature model $\mathcal{M}_x$, we need to remove the core property of feature model $\mathcal{M}_y$. Otherwise, there will be no opportunity to create a product, in which no feature of the instantiated feature model $\mathcal{M}_y$ is included. This is in contrast to the idea of feature-model instances, which should enable an optional inclusion of the instance. Therefore, we apply function R to the feature model $\mathcal{M}_y$ to remove this property. Afterwards, we can use function $\bullet$ to combine the feature models $\mathcal{M}_x$ and $\mathcal{M}_y$. First, we create the cross product of both feature models ($\times$, $\mathcal{M}_x \bullet \mathcal{R}(\mathcal{M}_y)$). Since not all resulting products are of our interest, we secondly use the function $\bullet$ to only result in products in which the additional (cross-tree) constraints of feature model $\mathcal{M}_c$ also hold ($\bowtie$, $(\ldots) \bullet \mathcal{M}_C$).

Running Example

Using our running example of the product line *BankApplication*, in which we want to reuse product line *BankAccount*, we now investigate the application of function $\circ$ (cf. Figure 4.1). In this scenario, we plan to instantiate the product line *BankAccount* below the feature *BankApplication* of the corresponding product line. Furthermore, we plan to add the cross-tree constraint *Transaction* $\rightarrow$ *SimpleLock* so that the feature *SimpleLock* is available in all products in which the feature *Transaction* is included. This ensures that the locking functionality can be used by the transaction feature.

For the instantiation of the feature model *BankAccount* and the additional cross-tree constraint, we create the feature model $\mathcal{M}_c$ with the set of features $\mathcal{F}_c$ and the set of products $\mathcal{P}_c$. In detail, the set of features $\mathcal{F}_c$ consists of the features *BankAccount* and *SimpleLock* from the product line *BankAccount*, and the features *BankApplication* and *Transaction* from the product line *BankApplication* ($\mathcal{F}_c = \{A, B, S, T\}$). These four features are needed to describe the dependencies between both product lines. However, as we defined a feature model with $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ whereas $\mathcal{F}$ is the set of features and $\mathcal{P}$ a set of products, we also need the set of products for the feature model $\mathcal{M}_c$. For

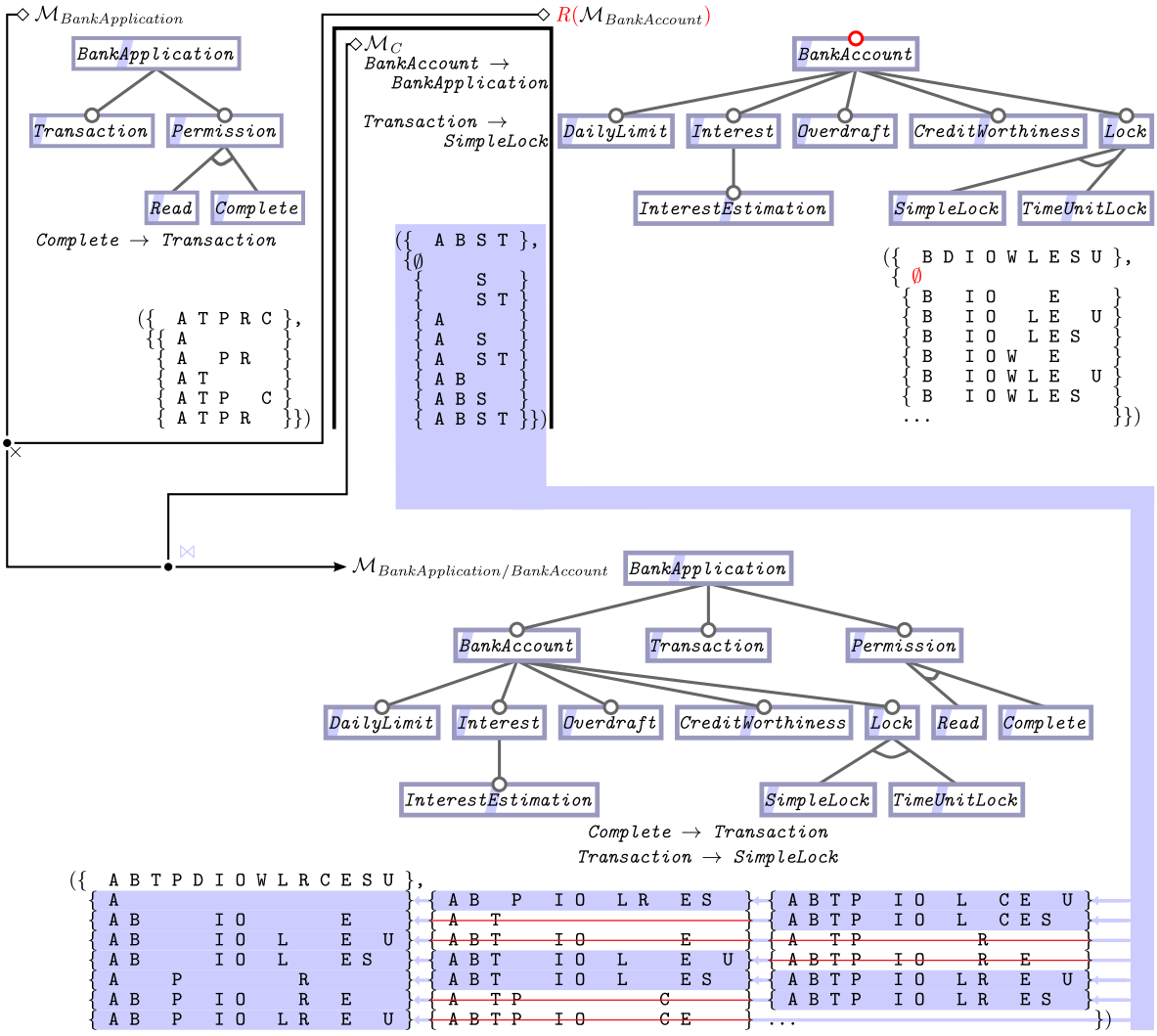


Figure 4.1: Feature-model composition using the feature models of the product lines *BankApplication* and *BankAccount* as input (according to [Schröter et al. 2016]). Feature model $\mathcal{M}_C$ is used to describe inter-model constraints.

this purpose, we have to convert the intended constraints into a product representation, such that $\mathcal{M}_c$ is completely defined and can be used as input for the function $\circ$. In detail, we convert the constraints to the propositional formula $B \rightarrow A \wedge T \rightarrow S$ and determine all valid feature combinations. As a result, we get the set of products $\mathcal{P}_c$ as depicted in Figure 4.1.

Using the feature models $\mathcal{M}_{BankApplication}$, $\mathcal{M}_{BankAccount}$, and $\mathcal{M}_c$, we collected all input values for the application of function $\circ$. First, we use the function R and apply it to feature model $\mathcal{M}_{BankAccount}$ to add the empty set to the set of valid products ($\mathcal{P}_{BankAccoupnt} \cup \{\emptyset\}$). Second, we create all product combinations of the feature models $\mathcal{M}_{BankApplication}$ and $\mathcal{M}_{BankAccount}$ with the empty set (i.e., $R(\mathcal{M}_{BankAccount})$) using the function $\bullet$ (i.e., as cross product $\times$). In Figure 4.1, we present the intermediate

result of this computation as feature model $\mathcal{M}_{BankApplication/BankAccount}$. Third, we use the function $\bullet$ with the intermediate result and the feature model $\mathcal{M}_c$ to remove all products of the feature model $\mathcal{M}_{BankApplication/BankAccount}$ that are not compatible to the constraints that we described using feature model $\mathcal{M}_c$ (see function $\bullet$, $\bowtie$). We depict the final feature model $\mathcal{M}_{BankApplication/BankAccount}$ in Figure 4.1 and highlight the set of resulting products.

4.1.2 Concept of Feature-Model Interfaces

Now, we define the feature-model interface as first interface of our concept for multi-level interfaces. As all interfaces of our concept, the feature-model interface also aims to remove the direct dependencies between dependent product lines and focuses on the artifacts of interest (i.e., features). As a result, it should be possible to use the interface instead of the original feature model for the purpose of modeling and analyses in a multi-product-line scenario (i.e., in combination with feature-model composition). Thus, the interface acts as a placeholder and we use the hidden product line in a productive system.

During the investigation of our ideas regarding the concept of feature-model interfaces, the question arises how to create such an interface. Since the feature-model interface should be exchangeable with the original product line (i.e., like a placeholder), a simple deletion of a feature from the feature model is not possible. For instance, if we consider the feature diagram for product line *BankAccount* and we want to remove feature *Interest*, it is not clear how to connect the remaining feature *InterestEstimation* with the root feature. Furthermore, if we take a look at the representation of a feature model as propositional formula the situation is also not clear [Krieter et al. 2016a]. Considering the propositional formula $R \wedge (A \rightarrow R) \wedge (B \rightarrow A)$ that represents a small feature model, in which R is the root feature, A is a subfeature of R, and B a subfeature of A. In this propositional formula, we want to remove feature B but a syntactical removal leads to wrong results. In detail, a syntactical removal leads to a representation, in which only the product $\{R, A\}$ exists and, thus, the product $\{R\}$ is not represented anymore.

By contrast to the previous consideration, we use the ideas presented with abstract features [Thüm et al. 2011a] and feature-model slicing [Acher et al. 2011] to create a feature-model interface with a reduced set of features and stable feature dependencies. Based on this, we show how to use the resulting feature-model interface in combination with feature-model composition to achieve benefits for the feature-model analysis. Therefore, we prove a set of analysis-result relations between a feature-model composition based on the feature-model interface and a composition based on the original feature model.

For the purpose of illustration, we take a look at our running example and define the feature-model interface afterwards. In detail, we plan to reuse the product line *BankAccount* inside of the product line *BankApplication*. In this context, we are interested in several features of product line *BankAccount* because they are needed to fulfill the functionality of the product line *BankApplication* or they are of our special interest due to the represented functionality. In detail, we are interested in the features *BankAccount*, *Interest*, *SimpleLock*, and *DailyLimit* (cf. Figure 3.2). Thus, we plan to remove all other features to create our feature-model interface. However, at the same time we need a feature-model interface in which the represented products are also compatible to the original feature model. This means, all combinations of features that represent a product in the feature-model interface also need to represent a product in the original feature model. For this purpose, we define the feature-model interface as follows:

Definition 9. Feature-model interface according to *Definition 4* of [Schröter et al. 2016]**:**

A feature model $\mathcal{M}_{Int} = (\mathcal{F}_{Int}, \mathcal{P}_{Int})$ *is an interface of feature model* $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ *denoted as* $\mathcal{M}_{Int} \preceq \mathcal{M}_x$*, if and only if the following conditions hold:*

(a) $\mathcal{F}_{Int} \subseteq \mathcal{F}_x$ *and*

(b) $\mathcal{P}_{Int} = \{p \cap \mathcal{F}_{Int} \mid p \in \mathcal{P}_x\}$.

Resulting from the definition of a feature-model interface, the feature-model interface is itself a feature model. Therefore, we can use this interface as a placeholder for the original feature model in a feature-model composition. Furthermore, the definition leads to some properties that are relevant for our proofs in the remaining thesis. First, for each feature model $\mathcal{M}_x$ and a set of features that represent the features of interest, it exists exactly one feature-model interface (i.e., it exists exactly one semantic representation even if there are multiple graphical representations as feature diagram [Czarnecki and Wąsowski 2007]). Second, each product of the feature-model interface $\mathcal{M}_{Int}$ is a subset (not necessarily a strict subset) of a product from the original feature model and contains only features from $\mathcal{F}_{Int}$. Third, the other way around also holds. This means, each product in the feature model $\mathcal{M}_x$ is a super set of at least one product in feature model $\mathcal{M}_{Int}$. In Corollary 10, we summarize these properties as follows:

Corollary 10. According to *Corollary 5* of [Schröter et al. 2016]**:**

$$\forall p \in \mathcal{P}_{Int} \exists q \in \mathcal{P}_x : p = q \cap \mathcal{F}_{Int}$$
$$\forall q \in \mathcal{P}_x \exists p \in \mathcal{P}_{Int} : p = q \cap \mathcal{F}_{Int}$$

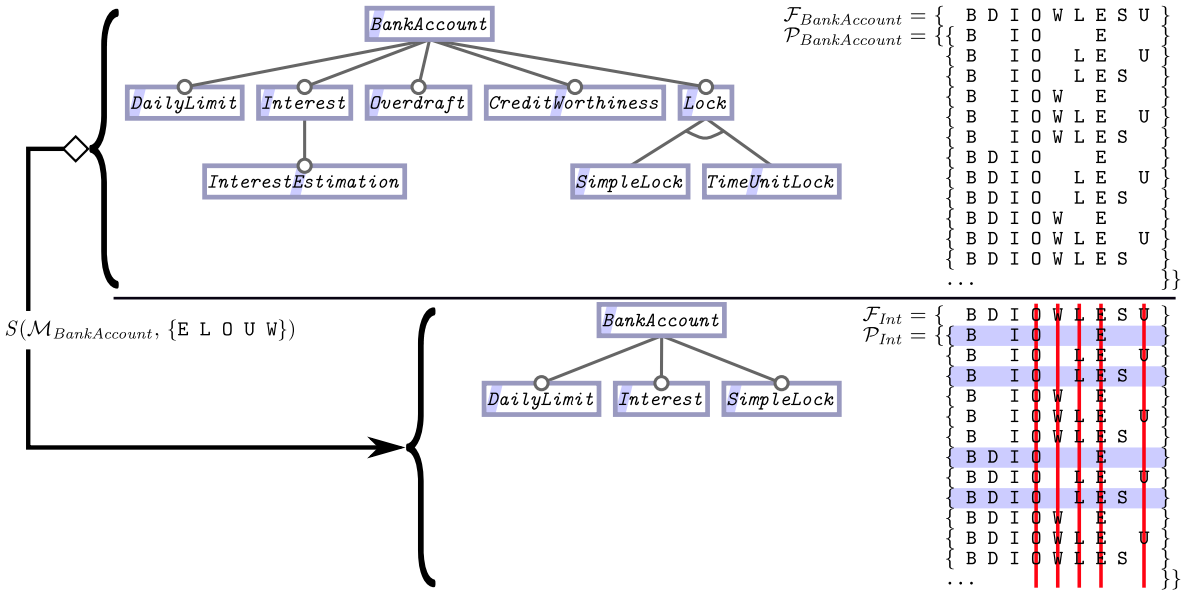


Figure 4.2: Application of function S using the feature model $\mathcal{M}_{BankAccount}$ and the features E, L, O, U, W as input. (The formal definition of the feature model uses level order for the representation of features).

Besides our definition of the feature-model interface and the subsequent properties, we also need to define a function that allows us to generate a feature-model interface based on an input feature model. In detail, we define function S (slice, which is similar to the respective operator proposed by Acher et al. [Acher et al. 2011]) as base for our proofs regarding the comparison of analysis-result relations between feature-model compositions with and without the feature-model interfaces. For this purpose, we define the function S that removes all features that are not of our interest from an input feature model:

Definition 11. Function S according to the *Definition 6* of [Schröter et al. 2016]**:**

We define a function S that takes a feature model $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ and a set of features $\mathcal{F}_R$ as input and returns a feature model $\mathcal{M}_{Int}$ with $\mathcal{M}_{Int} \preceq \mathcal{M}_x$.

$$\mathcal{M}_{Int} = S(\mathcal{M}_x, \mathcal{F}_R) = (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\})$$

Running Example

To exemplify the application of function S, we use our product line *BankAccount* with the feature model $\mathcal{M}_{BankAccount}$. As we are interested in the features *BankAccount* (B), *DailyLimit* (D), *Interest* (I), and *SimpleLock* (S), we initialize the set of features $\mathcal{F}_R$ with the complementary set of features according to $\mathcal{F}_{BankAccount}$. In detail, we define the set of features that we want to remove with $\mathcal{F}_R = \mathcal{F}_{BankAccount} \setminus \{B, D, I, S\} = \{E, L, O, U, W\}$. We depict the result of the application of function S with the described input values in Figure 4.2.

4.1.3 An Overview of Algebraic Properties

In this section, we investigate multiple properties of function S and prove their correctness. In detail, we take a look at the interactions of function S with function $\bullet$, and function R. Furthermore, we prove that the set of features $\mathcal{F}_R$ can act as a right identity element for function S.

Right Identity Element $\mathcal{F}_R$

First of all, we prove that the set of features $\mathcal{F}_R$ acts as a right identity element if $\mathcal{F}_R$ shares no features with the input feature model $\mathcal{M}_x$ (i.e., $\mathcal{F}_x \cap \mathcal{F}_R = \emptyset$). As a consequence, the input and output feature model will be identical ($S(\mathcal{M}_x, \mathcal{F}_R) = \mathcal{M}_x$).

Lemma 12. Right identity element according to *Lemma 7* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{F}_R$ a set of features with $\mathcal{F}_x \cap \mathcal{F}_R = \emptyset$, then $S(\mathcal{M}_x, \mathcal{F}_R) = \mathcal{M}_x$.

Proof.

As the intersection of $\mathcal{F}_x$ and $\mathcal{F}_R$ is the empty set, there will be no feature that is removed from the set $\mathcal{F}_x$. The result is the identical feature set $\mathcal{F}_x$. Similarly, the intersection between each product and the set of features $\mathcal{F}_R$ is also empty and, thus, each product will be the same as before.

$$S((\mathcal{F}_x, \mathcal{P}_x), \mathcal{F}_R) = ((\mathcal{F}_x \setminus \mathcal{F}_R), \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\}) \tag{12.1}$$

$$= (\mathcal{F}_x, \mathcal{P}_x) = \mathcal{M}_x \tag{12.2}$$

□

Distributivity of the Functions S and $\bullet$

In our second proof, we investigate the property of distributivity when using function S in combination with function $\bullet$. As a consequence, it does not matter in which order we apply both functions as the result will be identical.

Lemma 13. Distributivity of the functions S and $\bullet$ according to *Lemma 8* of [Schröter et al. 2016][2]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$ be feature models and $\mathcal{F}_R$ a set of features with $\mathcal{F}_x \cap \mathcal{F}_y \cap \mathcal{F}_R = \emptyset$, then

$$S(\mathcal{M}_x \bullet \mathcal{M}_y, \mathcal{F}_R) = S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(\mathcal{M}_y, \mathcal{F}_R).$$

[2]The Lemma 13 and the *Lemma 8* of [Schröter et al. 2016] differ in one important detail. Because of some advice from readers, we added a missing premise. Thank you for the feedback.

Proof.

In general, we separate the application of the function S on each part of the composed feature model so that we can apply function $\bullet$ later on.

$$S((\mathcal{F}_x, \mathcal{P}_x) \bullet (\mathcal{F}_y, \mathcal{P}_y), \mathcal{F}_R) \tag{13.1}$$
$$= (\mathcal{F}_z, \mathcal{P}_z) \tag{13.2}$$
$$= ((\mathcal{F}_x \cup \mathcal{F}_y) \setminus \mathcal{F}_R, \mathcal{P}_z) \tag{13.3}$$
$$= ((\mathcal{F}_x \setminus \mathcal{F}_R) \cup (\mathcal{F}_y \setminus \mathcal{F}_R), \mathcal{P}_z) \tag{13.4}$$

Next, without loss of generality, we introduce the sets r and s to represent the results of function S, which are then used as input for function $\bullet$.

$$= (\mathcal{F}_z, \{(p \cup q) \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \tag{13.5}$$
$$= (\mathcal{F}_z, \{(p \setminus \mathcal{F}_R) \cup (q \setminus \mathcal{F}_R) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \tag{13.6}$$
$$\overset{\text{(Definition 9)}}{=} (\mathcal{F}_z, \{r \cup s \mid r \in \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\}, s \in \{q \setminus \mathcal{F}_R \mid q \in \mathcal{P}_y\}, r \cap \mathcal{F}_y = s \cap \mathcal{F}_x\}) \tag{13.7}$$
$$\overset{\text{(Definition 8)}}{=} S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(\mathcal{M}_y, \mathcal{F}_R) \tag{13.8}$$ □

Distributivity of the Functions S and R

The third property focuses on the distributivity of function S and function R. Similar to the previous property, this leads to the consequence that the order in which we apply both functions does not influence the result.

Lemma 14. Distributivity of the functions S and R according to *Lemma 9* of [Schröter et al. 2016]**:**

Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{F}_R$ a set of features, then $S(R(\mathcal{M}_x), \mathcal{F}_R) = R(S(\mathcal{M}_x, \mathcal{F}_R))$.

Proof.

Function R adds the empty set to the set of products. To prove the interaction of R with S, it is necessary to extract this empty set from the input feature model of S.

$$S(R(\mathcal{M}_x), \mathcal{F}_R) = (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in (\mathcal{P}_x \cup \{\emptyset\})\}) \tag{14.1}$$
$$= (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\} \cup \{\emptyset\}\}) \tag{14.2}$$
$$= R((\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\})) \tag{14.3}$$
$$= R(S(\mathcal{M}_x, \mathcal{F}_R)) \tag{14.4}$$ □

4.1.4 Interface Dependencies of Feature-Model Compositions

In this section, we consider the interface dependency between feature models and investigate the interaction between the functions S and $\circ$. An interface dependency between two feature models exists, if we can apply function S with a specific set of features $\mathcal{F}_R$ on the first feature model to achieve the second feature model. Based on this, we prove that a feature-model composition based on a feature-model interface has also an interface dependency to a feature-model composition that is based on the feature model from which the feature-model interface was created. Especially this interface dependency is essential for our proofs of analysis-result relations of multiple analyses presented in the next section.

Before we start to prove the described dependency, we take a look at a more concrete example. Therefore, we assume that we plan to reuse parts of a feature model $\mathcal{M}_y$ in a second feature model $\mathcal{M}_x$. Without the proposed interface concept, we would use the function $\circ$ to combine both feature models accordingly ($\mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y = \mathcal{M}_{x/y}$). However, using the concept of feature-model interfaces, we can focus on the features of interest during this composition. Therefore, we create a feature-model interface $\mathcal{M}_{Int}$ that only consists of the features of interest. As both feature models depend on each other, we consider the relation as interface dependency (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_y$). Afterwards, we can use the resulting feature-model interface ($\mathcal{M}_{Int}$) for our feature-model composition with feature model $\mathcal{M}_x$ ($\mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int} = \mathcal{M}_{x/Int}$). The resulting feature-model composition $\mathcal{M}_{x/Int}$ is again a feature-model interface of the composition with the original feature model $\mathcal{M}_y$ (i.e., $\mathcal{M}_{x/Int} \preceq \mathcal{M}_{x/y}$). Therefore, the feature-model interface $\mathcal{M}_{x/Int}$ has the same properties as an ordinary feature-model interface that is not based on a composition.

In Lemma 15, we prove the described dependency:

Lemma 15. Interface dependency of feature-model compositions according to *Lemma 10* of [Schröter et al. 2016] **:**

Let $\mathcal{M}_{x/y} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y$, $\mathcal{M}_{x/Int} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int}$ *be composed feature models based on the models* $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$, $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R)$ *with* $\mathcal{F}_R \cap \mathcal{F}_x = \mathcal{F}_R \cap \mathcal{F}_C = \emptyset$, *then:* $\mathcal{M}_{x/Int} \preceq \mathcal{M}_{x/y}$.

Proof.

Given the algebraic properties of the function S and the definition of our composition function $\circ_{\mathcal{M}_C}$, the following relations hold:

$$\mathcal{M}_{x/y} \succeq S(\mathcal{M}_{x/y}, \mathcal{F}_R) \tag{15.1}$$
$$= S(\mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y, \mathcal{F}_R) \tag{15.2}$$
$$\overset{\text{(Equation 8.1)}}{=} S((\mathcal{M}_x \bullet R(\mathcal{M}_y)) \bullet \mathcal{M}_C, \mathcal{F}_R) \tag{15.3}$$
$$\overset{\text{(Lemma 13)}}{=} (S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(R(\mathcal{M}_y), \mathcal{F}_R)) \bullet S(\mathcal{M}_C, \mathcal{F}_R) \tag{15.4}$$
$$\overset{\text{(Lemma 12)}}{=} (\mathcal{M}_x \bullet S(R(\mathcal{M}_y), \mathcal{F}_R)) \bullet \mathcal{M}_C \tag{15.5}$$
$$\overset{\text{(Lemma 14)}}{=} (\mathcal{M}_x \bullet R(S(\mathcal{M}_y, \mathcal{F}_R))) \bullet \mathcal{M}_C \tag{15.6}$$
$$\overset{\text{(Definition 11)}}{=} (\mathcal{M}_x \bullet R(\mathcal{M}_{Int}))) \bullet \mathcal{M}_C \tag{15.7}$$
$$\overset{\text{(Equation 8.1)}}{=} \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int} \tag{15.8}$$
$$= \mathcal{M}_{x/Int} \tag{15.9}$$ □

Running Example

To illustrate the interface dependency in a feature-model composition, we use our running example of feature model *BankAccount* and feature model *BankApplication*. First of all, we can create the feature-model interface $\mathcal{M}_{Int}$ using the function S with the feature model $\mathcal{M}_{BankAccount}$ and the complementary set of features of interest (i.e., complementary set of *BankAccount*, *DailyLimit*, *Interest*, and *SimpleLock*) as input. As a result, an interface dependency between the feature-model interface $\mathcal{M}_{Int}$ and the input feature model $\mathcal{M}_{BankAccount}$ exists (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_{BankAccount}$). We depict this dependency in the upper part of Figure 4.3. Afterwards, we can use the original feature model $\mathcal{M}_{BankAccount}$ or the feature-model interface $\mathcal{M}_{Int}$ with feature model $\mathcal{M}_{BankApplication}$ as input for the feature-model composition. If we consider the resulting feature models, it also exists an interface dependency between the composed feature models (i.e., $\mathcal{M}_{BankApplication/Int} \preceq \mathcal{M}_{BankApplication/BankAccount}$, cf. lower part of Figure 4.3). However, according to Lemma 15, the application of function S on the composed feature model $\mathcal{M}_{BankApplication/BankAccount}$ has the same effect as an application of function S on the feature model $\mathcal{M}_{BankAccount}$ and a subsequent feature-model composition with feature model $\mathcal{M}_{BankApplication}$ (i.e., we assume the same features of interest).

4.2 Relation of Analysis Results With and Without Interfaces

In this section, we investigate the theoretical potential using feature-model interfaces to ease the analysis of feature-model compositions. For this purpose, we prove analysis-result relations of different feature-model analyses by comparing analysis results of composed feature models with and without feature-model interfaces (i.e., we assume an interface dependency between the composed feature models). In detail, we prove the analysis-result relations of the analyses *void feature model*, *core features*, *dead features*,

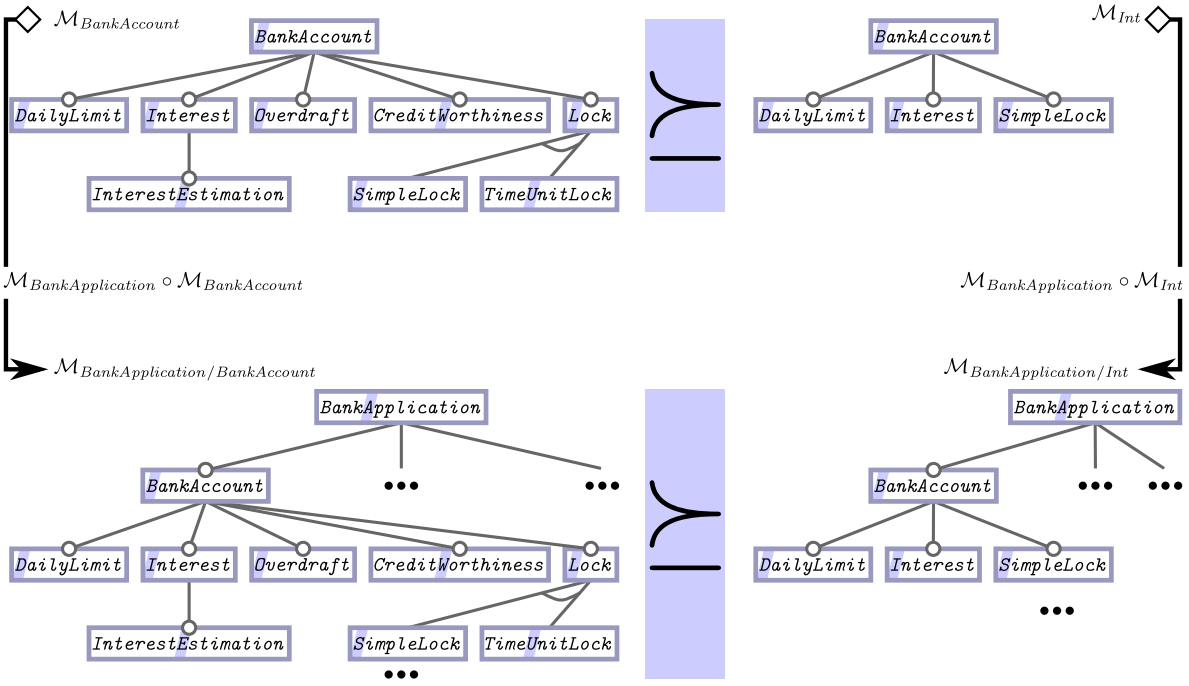


Figure 4.3: Interface dependency of feature models and feature-model compositions according to Lemma 15.

valid partial configurations, and *atomic sets* (see also Section 2.1.1, paragraph *Feature-Model Analyses*).

The subsequent proofs are based on the presented properties of the Section 4.1.3 and Section 4.1.4. As a result of these investigations, we consider a feature-model composition based on a feature-model interface as a new ordinary feature-model interface. In detail, using Lemma 15, we know that it is sufficient to prove analysis-result relations between a feature model $\mathcal{M}_y$ and a feature model $\mathcal{M}_{Int}$ as the same relations also hold for a composition based on these feature models (i.e., the relations also hold for feature model $\mathcal{M}_{x/Int}$ and feature model $\mathcal{M}_{x/y}$). However, we will prove these relations in detail and, therefore, we use the following premises:

Premise 1. According to *Premise 1* of [Schröter et al. 2016]**:**
Let $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$ be a feature model and $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R) = (\mathcal{F}_{Int}, \mathcal{P}_{Int})$ its feature-model interface (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_y$).

Premise 2. According to *Premise 2* of [Schröter et al. 2016]**:**
Let $\mathcal{M}_{x/y} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y$, $\mathcal{M}_{x/Int} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int}$ be composed feature models based on the feature models $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$, $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R)$ with $\mathcal{F}_R \cap \mathcal{F}_x = \mathcal{F}_R \cap \mathcal{F}_C = \emptyset$.

4.2.1 Void Feature Model

In our first proof, we consider the analysis of *void feature models* that is used to determine whether a feature model describes at least one product (cf. Section 2.1.1, Defini-

tion 2). In detail, we prove the analysis-result relation that a feature-model interface is void iff the corresponding feature model with an interface dependency is also void.

In Theorem 16 based on the Premise 1, we prove that a feature model $\mathcal{M}_{Int}$ is a void feature model if and only if the feature model $\mathcal{M}_y$ is a void feature model.

Theorem 16. Void feature model analysis-result relation according to *Theorem 11* of [Schröter et al. 2016]**:**

$$\mathcal{M}_y \in void \Leftrightarrow \mathcal{M}_{Int} \in void$$

Proof.

With Corollary 10, the following equivalences hold:

$$\mathcal{M}_y \in void \Leftrightarrow \mathcal{P}_y = \emptyset \tag{16.1}$$

$$\overset{(Corollary\ 10)}{\Leftrightarrow} \mathcal{P}_{Int} = \emptyset \tag{16.2}$$

$$\Leftrightarrow \mathcal{M}_{Int} \in void \tag{16.3}$$ □

In our second proof, we use the knowledge of Theorem 16 to prove that the same analysis-result relation also holds for feature-model compositions with and without a feature-model interface. To prove this relation, we also use the property of Lemma 15 in which we proved that a feature-model composition based on a feature-model interface also results in a feature-model interface compared to the feature-model composition with the corresponding feature model.

We prove the described relation of the composed feature models $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$ in Theorem 17 that is based on Premise 2.

Theorem 17. Void feature model analysis-result relation for composed feature models according to *Theorem 12* of [Schröter et al. 2016]**:**

$$\mathcal{M}_{x/y} \in void \Leftrightarrow \mathcal{M}_{x/Int} \in void$$

Proof.

From Lemma 15 and Theorem 16, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ *and* $\mathcal{M}_{x/y}$*.* □

4.2.2 Core Features

Similar to the analysis-result relations of *void feature models*, we now prove such relations for the analysis *core features*. As described in Section 2.1.1 using Definition 3, a core feature is a feature that is available in each product of the product line [Benavides et al. 2010; Trinidad and Ruiz-Cortés 2009]. However, we prove that a feature of a feature-model interface is a core feature iff this feature is a core feature of a corresponding feature model with an interface dependency.

In Theorem 18, we prove based on Premise 1 that a feature $f \in \mathcal{F}_{Int}$ is a core feature of a feature-model interface $\mathcal{M}_{Int}$ if and only if the feature is a core feature of feature model $\mathcal{M}_y$.

Theorem 18. Core features analysis-result relation according to *Theorem 13* of [Schröter et al. 2016]**:**

$$core(\mathcal{M}_y) \cap \mathcal{F}_{Int} = core(\mathcal{M}_{Int})$$

Proof.

With Definition 9, the following equation holds:

$$core(\mathcal{M}_{Int}) = \bigcap_{p \in \mathcal{P}_{Int}} p \tag{18.1}$$

$$\overset{(Definition\ 9)}{=} \bigcap_{p' \in \mathcal{P}_y} (p' \cap \mathcal{F}_{Int}) \tag{18.2}$$

$$= (\bigcap_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int} \tag{18.3}$$

$$= core(\mathcal{M}_y) \cap \mathcal{F}_{Int} \tag{18.4}$$ □

Using Theorem 18, we can conclude that a core feature of a feature-model interface $\mathcal{M}_{Int}$ is also a core feature of the feature model $\mathcal{M}_y$. Furthermore, the other direction also holds. Thus, if a feature f is a core feature of the feature model $\mathcal{M}_y$ and at the same time a feature of the feature-model interface, then the feature f is also a core feature of this interface. This conclusion can be summarized as follows [Schröter et al. 2016]:

$$f \in core(\mathcal{M}_{Int}) \Rightarrow f \in core(\mathcal{M}_y)$$
$$f \in core(\mathcal{M}_y) \cap \mathcal{F}_{Int} \Rightarrow f \in core(\mathcal{M}_{Int})$$

The second proof for the analysis core features also considers analysis-result relations of feature-model compositions with and without a feature-model interface. For this purpose, we use the knowledge of Theorem 18 and the property given in Lemma 15 to prove that this relation also holds for feature-model compositions.

In Theorem 19, we prove the analysis-result relation for the feature-model compositions $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$ using Premise 2.

Theorem 19. Core features analysis-result relation for composed feature models according to *Theorem 14* of [Schröter et al. 2016]**:**

$$core(\mathcal{M}_{x/y}) \cap \mathcal{F}_{x/Int} = core(\mathcal{M}_{x/Int})$$

Proof.

From Lemma 15 and Theorem 18, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$. □

4.2.3 Dead Features

In this section, we consider the analysis-result relation for the analysis *dead features* according to the Definition 4 of Section 2.1.1. To repeat, a dead feature is a feature that is not included in any product of the product line [Benavides et al. 2010; Kang et al. 1990]. Thus, we prove that a dead feature of a feature-model interface is also a dead feature of the corresponding feature model and that also the other way around holds if an interface dependency exists.

In compliance with Premise 1, we prove in Theorem 20 that a feature $f \in \mathcal{F}_{Int}$ is a dead feature of the feature-model interface $\mathcal{M}_{Int}$ iff the feature f is a dead feature of feature model $\mathcal{M}_y$.

Theorem 20. Dead features analysis-result relation according to *Theorem 15* of [Schröter et al. 2016]**:**

$$dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} = dead(\mathcal{M}_{Int})$$

Proof.

Based on Definition 9, the following equations hold:

$$dead(\mathcal{M}_{Int}) = \mathcal{F}_{Int} \setminus \bigcup_{p \in \mathcal{P}_{Int}} p \tag{20.1}$$

$$\overset{(Definition\ 9)}{=} (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus (\bigcup_{p' \in \mathcal{P}_y} (p' \cap \mathcal{F}_{Int})) \tag{20.2}$$

$$= (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus ((\bigcup_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int}) \tag{20.3}$$

$$= (\mathcal{F}_y \setminus \bigcup_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int} \tag{20.4}$$

$$= dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} \tag{20.5}$$ □

As conclusion of Theorem 20, we know that a feature $f \in \mathcal{F}_{Int}$ is a dead feature of the feature model $\mathcal{M}_y$ if the feature is a dead feature of the feature-model interface $\mathcal{M}_{Int}$. Furthermore, we also conclude that all dead features of the feature model $\mathcal{M}_y$ that are also part of the feature-model interface $\mathcal{M}_{Int}$ are also dead features of the feature-model interface. We summarize this conclusion as follows [Schröter et al. 2016]:

$$f \in dead(\mathcal{M}_{Int}) \Rightarrow f \in dead(\mathcal{M}_y)$$
$$f \in dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} \Rightarrow f \in dead(\mathcal{M}_{Int})$$

Similar to the previous analysis investigation, we take a look at the analysis-result relation with respect to feature-model compositions. For this purpose, we use the knowledge of Theorem 20 and Lemma 15 to prove that the same analysis-result relation also holds for a feature-model composition with and without a feature-model interface.

Based on Premise 2, we prove in Theorem 21 that the analysis-result relation also holds for the results of the feature-model compositions $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$.

Theorem 21. Dead features analysis-result relation for composed feature models according to *Theorem 16* of [Schröter et al. 2016]**:**

$$dead(\mathcal{M}_{x/y}) \cap \mathcal{F}_{x/Int} = dead(\mathcal{M}_{x/Int})$$

Proof.

From Lemma 15 and Theorem 20, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ *and* $\mathcal{M}_{x/y}$*.* □

4.2.4 Valid Partial Configurations

In this section, we investigate the analysis-result relation of the analysis *partial configuration*. In Section 2.1.1 with Definition 5, we defined a partial configuration as a tuple of selected and deselected features that is conform to the feature-model dependencies and, thus, valid. According to this definition, we prove that a valid partial configuration of the feature-model interface is also a valid partial configuration of a feature model $\mathcal{M}_y$ and the other way around if $\mathcal{M}_y$ is compatible to this interface.

Based on Premise 1, we prove in Theorem 22 that a configuration $C = (\mathcal{F}_S,\ \mathcal{F}_D)$ with $\mathcal{F}_S \subseteq \mathcal{F}_{Int}$ and $\mathcal{F}_D \subseteq \mathcal{F}_{Int}$ is a valid partial configuration of the feature-model interface $\mathcal{M}_{Int}$ if and only if C is a valid partial configuration of feature model $\mathcal{M}_y$.

Theorem 22. Partial configurations analysis-result relation according to *Theorem 17* of [Schröter et al. 2016]**:**

$$pConf(\mathcal{M}_{Int}) = \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)\}$$

Proof.

With Definition 9, the following equation holds:

$$\begin{aligned}
&pConf(\mathcal{M}_{Int}) \\
&\overset{(Definition\ 5)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists p \in \mathcal{P}_{Int} : \mathcal{F}_S \subseteq p \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_{Int} \setminus p\} && (22.1)\\
&\overset{(Corollary\ 10)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \\
&\qquad \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \ \wedge \\
&\qquad \mathcal{F}_D \subseteq \mathcal{F}_{Int} \setminus (q \cap \mathcal{F}_{Int})\} && (22.2)\\
&\overset{(\mathcal{F}_{Int} \subseteq \mathcal{F}_y)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \\
&\qquad \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \ \wedge \\
&\qquad \mathcal{F}_D \subseteq (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus (q \cap \mathcal{F}_{Int})\} && (22.3)\\
&= \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \\
&\qquad \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \ \wedge \\
&\qquad \mathcal{F}_D \subseteq (\mathcal{F}_y \setminus q) \cap \mathcal{F}_{Int}\} && (22.4)\\
&= \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid \exists q \in \mathcal{P}_y : \\
&\qquad \mathcal{F}_S \subseteq q \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_y \setminus q\} && (22.5)\\
&\overset{(Definition\ 5)}{=} \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)\} && (22.6)
\end{aligned}$$

□

Based on Theorem 22, we can conclude that all valid partial configurations of feature model $\mathcal{M}_{Int}$ are also valid partial configurations of feature model $\mathcal{M}_y$. Furthermore, we also know that an intersection of all selected features $\mathcal{F}_S$ and deselected features $\mathcal{F}_D$ with the set of features $\mathcal{F}_{Int}$ of all valid partial configurations of $\mathcal{M}_y$ also results in a valid partial configuration of feature model $\mathcal{M}_{Int}$. We summarize these conclusions as follows [Schröter et al. 2016]:

$$(\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_{Int}) \Rightarrow (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)$$
$$(\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y) \Rightarrow (\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \in pConf(\mathcal{M}_{Int})$$

Again, we use Theorem 22 and Lemma 15 to prove that the identical analysis-result relation of composed feature models with and without feature-model interfaces also exists.

In detail, we use Premise 2, to prove in Theorem 23 that a configuration with $\mathcal{F}_S \subseteq \mathcal{F}_{x/Int}$ and $\mathcal{F}_D \subseteq \mathcal{F}_{x/Int}$ is a valid configuration of the feature-model composition $\mathcal{M}_{x/Int}$ if and only if it is also a valid configuration of the feature-model composition $\mathcal{M}_{x/y}$.

Theorem 23. Partial configurations analysis-result relation for composed feature models according to *Theorem 18* of [Schröter et al. 2016]**:**

$$pConf(\mathcal{M}_{x/Int}) = \{(\mathcal{F}_S \cap \mathcal{F}_{x/Int}, \mathcal{F}_D \cap \mathcal{F}_{x/Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_{x/y})\}$$

Proof.

From Lemma 15 and Theorem 22, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$. □

4.2.5 Atomic Sets

Our last analysis-result relation considers the analysis of *atomic sets*. According to our Definition 6 of Section 2.1.1, an atomic set is a set of features that completely occurs or is completely absent in each product of a product line. For our proofs, we consider the analysis-result relation of an atomic subset which can also be a subset of another atomic (sub)set (cf. Definition 6). Therefore, we prove that an atomic subset of a feature-model interface is also an atomic subset of a feature model with an interface dependency.

For our Theorem 24, we base on Premise 1 and prove that a set of features $A \cap \mathcal{F}_{Int}$ with $A \subseteq \mathcal{F}_y$ is an atomic subset of the feature-model interface $\mathcal{M}_{Int}$ iff A is an atomic subset of the feature model $\mathcal{M}_y$.

Theorem 24. Atomic sets analysis-result relation according to *Theorem 19* of [Schröter et al. 2016]**:**

$$aSub(\mathcal{M}_{Int}) = \{q \cap \mathcal{F}_{Int} \mid q \in aSub(\mathcal{M}_y), q \cap \mathcal{F}_{Int} \neq \emptyset\}$$

Proof.

With Definition 9, the following equation holds:

$$
\begin{aligned}
aSub(\mathcal{M}_{Int}) &\overset{(Definition\ 6)}{=} \{q \mid q \neq \emptyset, \mathcal{P}_{Int} \neq \emptyset, \\
&\qquad \forall p \in \mathcal{P}_{Int} : (q \subseteq p) \vee (q \subseteq \mathcal{F}_{Int} \setminus p)\} && (24.1) \\
&\overset{(Corollary\ 10)}{=} \{q \mid \mathcal{P}_y \neq \emptyset,\ q \neq \emptyset, \\
&\qquad \forall p \in \mathcal{P}_y : (q \subseteq p \cap \mathcal{F}_{Int}) \vee \\
&\qquad\qquad (q \subseteq (\mathcal{F}_y \setminus p) \cap \mathcal{F}_{Int})\} && (24.2) \\
&= \{q \cap \mathcal{F}_{Int} \mid \mathcal{P}_y \neq \emptyset, q \neq \emptyset, q \cap \mathcal{F}_{Int} \neq \emptyset, \\
&\qquad \forall p \in \mathcal{P}_y : (q \subseteq p) \vee (q \subseteq \mathcal{F}_y \setminus p)\} && (24.3) \\
&\overset{(Definition\ 6)}{=} \{q \cap \mathcal{F}_{Int} \mid q \in aSub(\mathcal{M}_y), q \cap \mathcal{F}_{Int} \neq \emptyset\} && (24.4)
\end{aligned}
$$

□

As conclusion of Theorem 24, we know that an atomic set of the feature-model interface $\mathcal{M}_{Int}$ is also an atomic set (or a subset of an atomic set) of feature model $\mathcal{M}_y$. In addition, we also know that an atomic set of the feature model $\mathcal{M}_y$ intersected with $\mathcal{F}_{Int}$ also results in an atomic set of the feature-model interface $\mathcal{M}_{Int}$. We can summarize our conclusion as follows [Schröter et al. 2016]:

$$
\begin{aligned}
A \in aSub(\mathcal{M}_{Int}) &\Rightarrow A \in aSub(\mathcal{M}_y) \\
A \in aSub(\mathcal{M}_y) &\Rightarrow (A \cap \mathcal{F}_{Int}) \in aSub(\mathcal{M}_{Int})
\end{aligned}
$$

Using the knowledge of Theorem 24 and Lemma 15, we take a look into the analysis-result relation of composed feature models with and without a feature-model interface. Thus, we also prove that the relation of Theorem 24 also holds for feature-model compositions.

In Theorem 25 that is based on Premise 2, we prove that a set of features $A \cap \mathcal{F}_{x/Int}$ is an atomic subset with $A \subseteq \mathcal{F}_{x/y}$ of the feature model $\mathcal{M}_{x/Int}$ if and only if A is an atomic subset of the feature model $\mathcal{M}_{x/y}$.

Theorem 25. Atomic sets analysis-result relation for composed feature models according to *Theorem 20* of [Schröter et al. 2016]**:**

$$aSub(\mathcal{M}_{x/Int}) = \{q \cap \mathcal{F}_{x/Int} \mid q \in aSub(\mathcal{M}_{x/y}), q \cap \mathcal{F}_{x/Int} \neq \emptyset\}$$

Proof.

From Lemma 15 and Theorem 24, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ *and* $\mathcal{M}_{x/y}$*.* □

4.3 Evaluation: The Feature-Model Interfaces in Practice

In this section, we take a look at the practical application of feature-model interfaces. In detail, we investigate our Hypothesis 1 and explore how feature-model interfaces can be used to ease automated analyses of evolving multi product lines.

To investigate Hypothesis 1, first, we describe an application scenario and, second, we define research questions that we want to answer in our experiment.

Application Scenario

In our application scenario, we are interested in the benefits of our concept regarding the analysis of the feature-model composition $\mathcal{M}_{BankApplication}$ with the feature-model interface $\mathcal{M}_{Int}$ (i.e., $\mathcal{M}_{BankApplication/Int}$) that we created from feature model *BankAccount*. In addition, we assume that the feature model *BankAccount* changes several times. However, not all of these changes require a recomputation of a specific analysis as the new version of the feature model *BankAccount* can also be compatible to the already used feature-model interface in our feature-model composition (e.g., $\mathcal{M}_{Int} \preceq \mathcal{M}_{BankAccount,V2}$). Therefore, the question arises how often a new version of the feature model *BankAccount* is incompatible to the used feature-model interface $\mathcal{M}_{Int}$ (e.g., $\mathcal{M}_{Int} \not\preceq \mathcal{M}_{BankAccount,V<VersionNumber>}$). Only in this case, we have to create a new version of the feature-model interface $\mathcal{M}_{Int}$ and a new version of our feature-model composition with $\mathcal{M}_{BankApplication}$ that also leads to a necessary recomputation of all analyses (cf. *Case 2* of Section 3.5). In all other cases, in which the new version of the $\mathcal{M}_{BankAccount}$ is still compatible to the feature-model interface $\mathcal{M}_{Int}$ (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_{BankAccount,V<VersionNumber>}$), we can reduce computational effort as a recomputation would present the identical results.

Research Questions

According to the described application scenario, we now consider research questions to investigate our Hypothesis 1. In detail, we consider the research questions from our corresponding paper, in which we introduced the concept of feature-model interfaces [Schröter et al. 2016].

Research Question 1: How small can feature-model interfaces get compared to their corresponding feature models?

> This research question is directly related to our application scenario. In general, smaller feature models with less complexity are easier to analyze than huge feature models. Even if the process of automated analysis is in general very time efficient, the performance depends on the analysis and how often we need to execute this analysis. Therefore, even small analysis improvements can have an impact to specific scenarios. In addition, a smaller feature model with less complexity can ease the comprehension and the manual analysis by a developer. Therefore, we analyze the relation between the size of feature-model compositions with and without feature-model interfaces.

Research Question 2: How often does a feature-model interface become incompatible to an evolved feature model?

The result of this question directly influences the success of feature-model interfaces for our application scenario. In detail, we only reduce computational effort for feature-model compositions if the evolved feature model is still compatible to the corresponding feature-model interface (cf. *Case 1* of Section 3.5). Otherwise, we have to recompute the complete analysis using a new feature-model composition with a new feature-model interface that is compatible to the evolved feature model (cf. *Case 2* of Section 3.5).

Research Question 3: Is it possible to achieve performance benefits using compositional analysis for atomic sets compared to an analysis of the complete feature model?

Even if our Hypothesis 1 focuses on evolving multi product lines, we take a look at further potential of feature-model interfaces regarding the analysis of composed feature models. Therefore, we compare the feature-model analysis of atomic sets with and without feature-model interfaces whereas we are focusing on analysis results regarding our features of interest. To illustrate the scenario, we use our running example. Without feature-model interfaces, we assume to create a feature-model composition of multi product line $\mathcal{M}_{BankApplication/BankAccount}$. However, during the analysis of this feature model, we are only interested in results regarding our features of interest. Therefore, we apply a specific feature-model analysis on the whole feature model of the multi product line and filter the result according to our features of interest. By contrast, using our concept of feature-model interfaces with feature-model composition and the knowledge of the proved analysis-result dependencies, we can use the feature-model composition of $\mathcal{M}_{BankApplication/Int}$ as input for the specific analysis to achieve the same results. According to this research question, we want to investigate whether this procedure can result in performance benefits for the specific analysis.

In the following, we introduce the details of our experiment design including an introduction of the subject system. Afterwards, we discuss the results of our experiments regarding our application scenario and present threats to validity.

4.3.1 Experiment Design

In this section, we present the experiment design to answer the research questions defined in the last section. Before we give details about the design itself, we present facts of our subject system that we use to answer all research questions.

Subject System

An experiment according to the presented research questions has high requirements regarding the subject system. For instance, the system needs to consist of different feature models (i.e., one feature model that instantiates further feature models that we call submodels to ease the following description) for which we also need to know how to compose them. Furthermore, we need information about evolutionary changes of the

instantiated submodels to adjust our application scenario. For these reasons, it was hard to find an appropriate subject system. However, we found a subject system that in most cases fits our needs.

As subject, we use a real-world feature model that comes from the automotive domain describing hardware and software artifacts. We have access to four different snapshots of an obfuscated version of this feature model (i.e., each feature has a unique ID over all snapshots). Depending on the specific snapshot, the feature model consists of more than 40 submodels. For instance, the first snapshot consists of 14,010 features, 666 constraints, and 44 submodels. By contrast, our last snapshot has 18,616 features, 1,616 constraints, and 46 submodels. To reproduce our results and to allow other researchers to use these snapshots for their own interests, we make them available in FeatureIDE's example wizard [Meinicke et al. 2017; Thüm et al. 2014b].

Experiment

As described above, our subject system is one huge feature model from which we need the initial submodels for our experiment. For this purpose, we also get the information about the original root features of these submodels. This allows us to reproduce the original feature models (i.e., submodels) and to use them for our experiment. In detail, we execute the feature model extraction process in two steps. First, we search for the specific location of each root feature in the complete feature model and create a new feature model with a copy of this feature and all sub features (constraints are ignored for now). Second, we investigate the existing constraints of the complete feature model and classify each constraint to either an intra-model constraint or an inter-model constraint. Afterwards, we copy all intra-model constraints into the corresponding feature models (i.e., submodels). In addition, we save all inter-model constraints as we need them to recompose the complete feature model.

As our research questions focus on the evaluation of the size and compatibility of the submodels to the corresponding feature-model interfaces, we also need to generate feature-model interfaces. For this purpose, we have to identify the features of interest according to each submodel. As the complete feature model is presented in an obfuscated way, it is not possible to use domain knowledge for the identification of features of interest. For this reason, we use the information given by the inter-model constraints (including the parent-child relationships between the root features of the submodels and the root model) to identify the features that we want to include in the feature-model interfaces. In detail, the features of the inter-model constraints are the features that represent the minimal degree of a possible feature-model interface as these features are at least necessary to describe the dependencies between all submodels and the root model. Thus, we use the information about the inter-model constraints to generate the feature-model interfaces for each submodel and snapshot. Afterwards, we use the information to determine the relation of the feature-model size (i.e., RQ1) and whether the submodel of a new snapshot is compatible to the feature-model interface of previous snapshot (i.e., RQ2). In addition, we use the analysis of atomic sets, as it is an

analysis with an exponential complexity [Durán et al. 2017], to investigate our third research question. In detail, we execute the analysis with and without our feature-model interfaces and determine the time that is needed to identify all atomic sets regarding our reduced feature-model composition with the features of interest. First, we use the complete feature model of the first snapshot and determine all atomic sets and filter the results according to the features of interest. Second, as comparison, we determine the complete time that is needed to (1) create all interfaces of the submodels, (2) to recompose the reduced feature model using the feature-model interfaces, and (3) to apply the analysis of atomic sets.

For the generation of the feature-model interfaces, it is necessary to use a scalable algorithm that allows us to remove a high percentage of features from a feature model. In our test runs, we realized that our algorithm, which we primary used for the removal of abstract features [Thüm et al. 2011a], does not scale well for this purpose. Therefore, we analyzed the properties of the algorithm and found several options for an optimization. For instance, as the features are always removed stepwise, we use a heuristic to optimize the order. However, the details of this algorithm are out of the scope of this thesis. We refer the reader to the work of Krieter et al. to get more insights in the algorithm's details [Krieter et al. 2016a,b]. The algorithm itself is available in FeatureIDE v3.0 [Meinicke et al. 2017; Thüm et al. 2014b].

4.3.2 Experiment Results and Discussion

In this section, we present the results of the experiments according to our research questions.

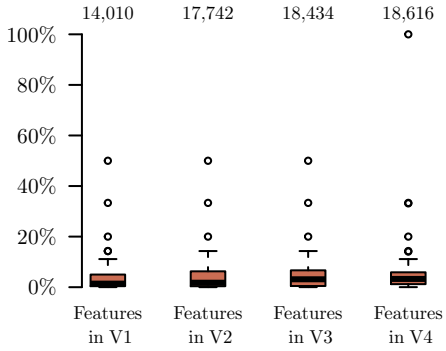


Figure 4.4: Each boxplot represents the percentage of features in the feature-model interface compared to the corresponding feature model of our subject system.

Results of Research Question 1

In Figure 4.4, we present the results of our evaluation regarding the first research question. In detail, we determine the percentage of features that exist in each feature-model interface with respect to the number of features in the corresponding feature model.

For instance, in the first snapshot exists one feature model (i.e., a submodel) with 7,800 features whereas the corresponding feature-model interface only consists of the root feature (i.e., the percentage of the number of features is less than 1). Furthermore, to ease the representation, we remove all submodels that only consist of the root feature. Afterwards, we use boxplots to represent these values for each snapshot.[3]

The first boxplot illustrates the results for the first snapshot of the automotive feature model. The median of this boxplot is below 1.4 % and, thus, we know that the half of all values are less than this value. Furthermore, we can see that further 25 % of the feature-model interfaces (i.e., upper border of the box) have also less or equal than 5 % of features relative to the corresponding feature model. The highest value for the first snapshot is about 50%. However, for all other snapshots, the relation between the number of features of the feature-model interface and the corresponding feature model looks similar. In detail, the median of all boxplots is less than 3.3 % and also the next 25 % of feature-model interfaces consist of less than 6.7 % of features. In sum, we can conclude that the difference between the size of a feature-model interface and the corresponding feature model is significant.

Results of Research Question 2

To answer the second research question, we investigate the compatibility of a new feature-model version to the feature-model interface of the previous snapshot (i.e., $\mathcal{M}_{Int,Vx} \preceq \mathcal{M}_{FM,Vx+1}$). We illustrate the results in Figure 4.5. In detail, if a submodel (i.e., feature model) changes, three different results are possible. First, the new feature-model version is compatible to the feature-model interface of the previous snapshot (brown). Second, the new feature-model version is incompatible (light green). Third, we find out that the feature-model version is incompatible because of missing domain knowledge (yellow). To be precise, in the third case (i.e., the yellow case), the feature-model interface is not compatible anymore since the features of interest changed although these features were also available in the previous snapshot. However, with domain knowledge it could be possible to know that these features are of our interest. As a result of this knowledge, the feature model would be compatible. Therefore, we differentiate this incompatibility with an additional case (yellow).

Our first bar of Figure 4.5 presents the compatibility from the first to the second snapshot. In sum, there are 44 of comparable feature models. However, out of these 44 submodels, 19 feature models have been changed. Therefore, we use these 19 feature models for the classification according to our three cases. As a result, 16 feature models are compatible (brown), one is incompatible (light green), and two are incompatible because of the missing domain knowledge (yellow). If we take a look at the other snapshots, the situation differs. The second bar represents the compatibility of feature-model interfaces from the second snapshot to feature models of the third snapshot. In this scenario, only 14 feature models have been changed, whereas 7 are still

[3]As we used *R* for the graphical representation, we also want to note that we used the default settings for the boxplots. Thus, the whiskers are equal or less than 1.5 times away from the box.

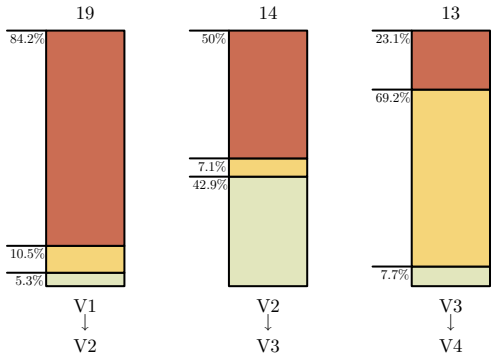


Figure 4.5: Each barplot represents the changes between two snapshots and shows the percentage of compatible interfaces (■), incompatible interface (minimality) (■), and incompatible interfaces (■) in our subject system.

compatible to the corresponding feature-model interface of the previous snapshot. Furthermore, 6 feature models are incompatible to the feature-model interface and only one is incompatible because of domain knowledge. Also the last bar that represents the compatibility of feature-model interfaces from the third snapshot to feature models of the fourth snapshot, looks completely different in comparison to the others. Here, only 13 feature models have been changed. Our investigation shows that 3 feature models are compatible (brown), only one is incompatible (light green), and 9 are incompatible because of missing domain knowledge (yellow). Using the information given by this bar, we can conclude that domain knowledge is very important to create a significant feature-model interface. In sum, our experiment shows that a new version of a feature model is compatible in more than the half of all cases. In addition, another 25 % of all feature models were only incompatible because of missing domain knowledge. Even if we assume that not all of these cases will lead to a compatible interface using domain knowledge, we suppose to increase the compatible cases.

Results of Research Question 3

In addition to our investigations regarding Hypothesis 1, we also examined the potential of feature-model interfaces with feature-model composition regarding the analysis of composed feature models. In detail, we focused on an analysis to determine all atomic sets of the reduced feature model of the first snapshot (i.e., we are only interested in atomic sets, in which the features of interest are considered). Using the complete feature-model composition as input for the analysis of atomic sets and a subsequent filtering according to the features of interest (the time of the filtering is negligible), the computation takes more than 50 hours. By contrast, the computation of atomic sets regarding the reduced feature model based on a feature-model composition with the feature-model interfaces, the computations take less than 5 seconds. This time includes the computation of the feature-model interface, the reconstruction of the reduced feature model, and the execution of the atomic-set analysis. Even if it is also possible to

optimize the internal behavior of the atomic-set algorithm, we decided to consider the algorithm as a black box. Since, an optimization is out of scope of our evaluation and we only focused on an investigation of our third research question. However, based on this comparison, we can confirm that feature-model interfaces with feature-model composition can lead to performance benefits when executing feature-model analyses.

General Conclusion

As described above, our application scenario focuses on the reduction of effort for automated analysis especially in the case of feature-model evolution (cf. Hypothesis 1). Using the results of our first research question, we can conclude that feature-model interfaces are able to significantly reduce the size of composed feature models. Thus, if we repeat an analysis several times (e.g., because of feature-model evolution, or a more complex analysis such as product-line type checking), we can also save computational effort for feature-model analyses that need in general less time. However, the main benefit of feature-model interfaces exists for feature-model evolution. In detail, using the results of our second research question, we can see that a new version of a feature model does not lead to a new feature-model interface in more than 50 % of all cases. Thus, it is not necessary to re-execute a specific analysis as the result will be the same as before. In addition to the investigation regarding the benefit of feature-model interfaces in evolved feature-model compositions, we also considered the potential of feature-model interfaces in cases of an ordinary feature-model analysis. Using the analysis of atomic sets, we were able to show that feature-model interfaces with feature-model composition can lead to performance benefits when we focus on results regarding our features of interest.

4.3.3 Threats to Validity

In this section, we discuss the threats to validity of our evaluation results. Again, we differentiate between the external an internal validity.

External Validity. To consider the external threats to validity, we take a look at the generalizability of our evaluation results. Indeed, our results strongly depend on our subject system from the automotive domain. In detail, it depends on our analyzed feature model with the information about the root features of the submodels. In addition, it also depends on the selected features of interest as we need this information to create the feature-model interfaces. Because of these dependencies, it is difficult to generalize the results or to predict the outcome of an evaluation regarding other multi product lines. Thus, it is necessary to extend the evaluation to further snapshots, feature models, and domains to generalize the results. However, for our evaluation we had no influence on the selected feature model and snapshots to minimize the effect of this threat to validity.

During the evaluation of the subject system, we got the impression that the feature model was extracted from a very early state of the development process. In detail, the first snapshot is based on 14 000 features, whereas the fourth snapshot is based

on 18 616 features. We assume that subsequent snapshots present results with less differences in features and their dependencies so that also the result of our evaluation differs. Therefore, further evaluations are needed to get more insights in the subject system and the impact on the compatibility of feature-model interfaces regarding their underlying submodels.

To create the feature-model interfaces, we used an automated extraction process, in which we analyzed the inter-model constraints of the feature-model composition to determine features of interest. This procedure leads to a minimal feature-model interface that only consists of features that are necessary to ensure the already existing inter-model constraints. One reason for this procedure was the missing domain knowledge that is necessary to create meaningful feature-model interfaces. Furthermore, we received the different versions of the feature model in an obfuscated way so that it was not possible for us to infer the meaning of a specific feature. As result, we considered the analysis of inter-model constraints as an appropriate procedure to find features of interest.

Internal Validity. According to the internal validity, we now consider our activities. To generate our feature-model interfaces, we used our prototypical implementation of an algorithm that was able to create the interfaces in a scalable manner. The prototypical implementation represents an internal threat because of possible undetected errors. Therefore, we used a state-of-the-art algorithm [Thüm et al. 2011a] to compare the results and to ensure the correctness. In detail, even if it was not possible to use this state-of-the-art algorithm to handle huge feature models as needed in our evaluation, we used a set of smaller feature models to compare the results of the algorithms and, thus to ensure the correctness of our prototypical implementation.

Furthermore, the choice of features of interest directly influences the size, representation, and compatibility of feature-model interfaces and, thus, also our evaluation result. As already described, we used an automated extraction process that leads to minimal feature-model interfaces. However, we are aware that a refactoring of cross-tree constraints is able to decrease the number of features of interest and the size feature-model interfaces. For instance, assuming the inter-model cross-tree constraint $A \rightarrow B \wedge C$ whereas the features A and B are part of the same and feature C from another feature model. Based on our extraction process, we can determine the features of interest for the feature model with the features A and B whereas both features are included. By contrast, using a refactoring, we can separate this inter-model constraint into the constraints $A \rightarrow B$ and $A \rightarrow C$. As a result, only the constraint $A \rightarrow C$ is now an inter-model constraint, whereas the constraint $A \rightarrow B$ represents an intra-model constraint. Thus, if we use our automated extraction process, only the feature A is a feature of our interest. Indeed, the approach with such a refactoring can lead to better evaluation results. Nevertheless, we made the decision that we use the extraction process without the refactoring, as it may harm the readability.

4.4 Summary

In this chapter, we introduced the feature-model interface as first interface of our multi-level interface concept. As we described, the feature-model interface is an ordinary feature model with a subset of features in relation to an already existing feature model that we planned to use in another product line. Therefore, the feature-model interface hides all other features that are not required in the context of the multi product line. However, the feature-model interface is the most important interface of our concept of multi-level interfaces as it also restricts the reuse and the visibility of the artifacts in the interfaces below.

According to Hypothesis 1, we investigated the potential of the feature-model interface (i.e., feature-model interface as representation of a variability-model interface) in theory and practice. For the theory part, we proved dependencies between analysis results of feature-model composition with and without a feature-model interface. As a result of these proofs, we found out that it is possible to exchange the feature-model interface with the original feature model (and vice versa) and it is not necessary to reanalyze the feature-model composition if only the features of interest are focused. Thus, in theory, we can use this property to reduce the computational effort during the automated analyses of composed feature models when using feature-model interfaces. In detail, we proved this relation for the analysis of void feature models, core features, dead features, partial configurations, and atomic sets.

Based on our proofs of analysis-result relations for feature-model compositions with and without feature-model interfaces, we also investigated how this can be used to safe computational effort during feature-model evolution in practice. Using a real-world feature model from the automotive domain, we investigated the advantages for the automated analyses regarding the feature-model compositions with feature-model interfaces. In detail, not all changes of a feature model, which we used behind a feature-model interface, forced a change of the feature-model interface. Thus, a re-execution of a specific analysis was not necessary as the result would be the same. However, in our real-world example, it was possible to prevent the re-execution of automated analysis in more than 50 % of all cases.

5. The Syntactical Interface - The Second Level of Multi-Level Interfaces

> This chapter shares material with the paper *Towards Modular Analysis of Multi Product Lines*, in which we introduced our first ideas of a variable interface as syntactical product-line interface [Schröter et al. 2013a]. Furthermore, the chapter shares material with the paper *Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines*, in which we described our main concept of feature-context interfaces [Schröter et al. 2014].

In this chapter, we focus on the syntactical interface of our multi-level interfaces to avoid direct dependencies on this level of the product-line development. It is the second interface of our concept and depends on the variability-model interface (or rather the feature-model interface) that we defined in the previous chapter. In detail, the syntactical interface is a collection of API members (i.e., methods and fields) with variability information that are available in the selected features of the variability-model interface (i.e., features of interest). Whereas the manual search for safely accessible API members for an implementation context can be a challenging task, the syntactical interface can be used to give an overview of it. To be more precise, the implementation context is defined by the product-line artifact that is currently implemented, changed, or maintained (e.g., a specific feature when using FOP for the implementation). Searching for accessible members of other implementation artifacts (e.g., features from another product line), which we can call from this implementation context to reuse its functionality, is tedious. By contrast, based on the syntactical interface, it is possible to create a non-variable interface through filtering techniques so that a developer gets an overview of all safely accessible members tailored to the current implementation context.

As the previous description of the syntactical interface is an abstract description that can be used for different languages and implementation techniques for product lines, we want to exemplify the strategy using one specific scenario. In detail, similar to the feature-model interface, we want to illustrate the concept using our running example of the multi product line *BankApplication* that uses product line *BankAccount*. Therefore, our interface for product line *BankAccount* depends on the information given by the feature-model interface (see Chapter 4), the programming language Java, and the implementation technique FOP (see Section 2.1.2). Focused on FOP and Java, we implemented the variable interface as an instance of the syntactical interface and the feature-context interface as an instance of the non-variable view to this interface. Thus, during the implementation of a specific feature for multi product line *BankApplication*, we assume that the feature-model interface, and the variable interface of product line *BankAccount* exist. For instance, if a developer wants to implement the feature *Transaction*, we can use the interfaces to automatically compute safely accessible API members that we can use for the development. In detail, the implementation context of this (multi) product line based on FOP is automatically given by the feature *Transaction* and we can use the information of the context, the feature-model interface, and the variable interface to compute the feature-context interface that presents a non-variable to all safely accessible API members. As result, the developer does not need the detailed information of product line *BankAccount* to implement features of product line *BankApplication*.

Even if the main idea of a variable interface and a subsequent feature-context interface is a result of the multi-level interface concept and focuses on multi product lines, the concept is also applicable to a single product line [Schröter et al. 2014]. This chapter focuses on both scenarios, in which the feature-context interface is used for the development of product lines and multi product lines. Therefore, the chapter is structured as follows. In Section 5.1, we start with a presentation of state-of-the-art strategies to identify accessible code members of a (multi) product line. In Section 5.2, we present our main idea and the generation strategy for a variable interface and discuss advantages and drawbacks for the product-line development. Afterwards, we discuss the application of the variable interface in different scenarios, such as multi product lines. To this end, in Section 5.3, we present filtering techniques for the variable interface to tailor it to the specific needs of each application scenario. Furthermore, we define feature-context interfaces in Section 5.4 and give details about their application in single as well as in multi product lines. Using several case studies, we also investigate the potential of feature-context interfaces as development support for (multi) product lines in Section 5.5. Finally, in Section 5.6, we summarize the chapter.

5.1 Finding Reusable Implementation Artifacts

Implementing product lines is an error-prone task as reusable artifacts (e.g., methods and fields) for the current implementation task need to be identified. For instance, if we use artifacts that are not available in all valid feature-combinations, the implementation can lead to compile-time errors. The task becomes even harder when considering multi

product lines as the systems complexity is increased. Therefore, we use this section to give an overview of state-of-the-art approaches that can be used to identify accessible implementation artifacts for product lines implemented with FOP. On the one hand, the considered approaches are based on our own experiences when implementing product lines. On the other hand, we consider straight-forward approaches that can be used for the identification.

To support the developer during an implementation or maintenance task of a product line, it is necessary to identify accessible members (i.e., methods and fields) according to an implementation context. In the case of a product line written with FOP, the implementation context is represented by the feature module that the developer implements, changes, or maintains. Thus, the implementation context corresponds to a single feature from the set of all features $\mathcal{F}$. Therefore, we call the implementation context for FOP feature context $\mathcal{FC}$, whereas $\mathcal{FC} \in \mathcal{F}$ holds.[4] Furthermore, to ease the comparison of the state-of-the-art approaches, we investigate whether the specific approach is applicable to the concept of multi product lines. Additionally, we introduce the criteria of *soundness* and *completeness* in Definition 26 to evaluate their usage in product lines and multi product lines.

Definition 26. Completeness and soundness according to [Schröter et al. 2014]**:**

- Completeness: *An approach is* complete *if it contains all members of an SPL that are safely accessible for each feature context.*
- Soundness: *An approach is* sound *if it contains only members that are safely accessible for each feature context.*

In detail, an unsound approach presents API members that can lead to compile-time errors. As a result, the developer cannot rely on the outcomes of an unsound approach. Furthermore, an incomplete approach presents not all members that are safely accessible. Therefore, an incomplete approach can restrict the reusability of accessible API members. In addition to these criteria that are mainly introduced to evaluate state-of-the-art approaches for product lines, we also want to discuss the applicability of each approach regarding its usage in the application scenario of multi product lines. In an optimal case, the approach should be complete, sound and applicable for multi product lines at the same time. In the following, we use these criteria to investigate approaches that we call *feature module*, *minimal variant*, and *always available*.

5.1.1 Approach 1: Feature Module

In practice, especially for students that are beginners in the area of product lines written in FOP, the feature module that is currently implemented is used to identify safely accessible members. This observation is based on our experience regarding our lectures on product lines. Here, students are often careful in using calls to API members of other

[4] If we take another implementation paradigm (cf. Section 2.1.2), the implementation context can be more complex. For instance, the implementation context of preprocessors can be an arbitrary propositional formula that is based on all defined features.

feature modules and start to search for an appropriate API member in the feature module that is currently implemented. Therefore, if we search for accessible API members according to the feature context $\mathcal{FC}$ with the approach *feature module*, we collect all API members of the corresponding feature module $impl(\mathcal{FC})$ (cf. Definition 7).

For instance, we assume that we plan to extend, maintain, or implement the feature *Transaction* of the product line *BankApplication* (cf. Figure 2.14). For this case, the feature context is represented by the feature *Transaction* (i.e., $\mathcal{FC} = Transaction$). Using the feature module as source to get accessible code artifacts (i.e., $impl(Transaction)$), we get the methods `lock` and `transfer` from the product line *BankApplication* as the result.

In sum, the approach *feature module* is sound as it only collects API members from the feature module that also represents the feature context. However, the approach *feature module* ignores all API members of the other feature modules, even if the usage of the feature module directly implies the other (e.g., a child feature implies the parent feature). For instance, if we want to implement feature *InterestEstimation* (cf. Figure 2.1), it is obvious that we can also access all API members of the parent feature *Interest* and of the root feature *BankAccount*. Furthermore, if we take a look at our application scenario of a multi product line, this approach does not help to detect accessible members of a reused product line. For instance, if we implement feature *Transaction*, the approach only collects API members of this feature from product line *BankApplication* and ignores all other features from this and the reused product lines.

5.1.2 Approach 2: Minimal Variant

The second approach uses a specific variant p of the product line to collect accessible API members according to a feature context (i.e., $impl(p)$, cf. Definition 7). The approach is similar to an existing implementation of FeatureIDE [Meinicke et al. 2017; Thüm et al. 2014b]. In detail, FeatureIDE uses the currently active configuration to create a variant in the background of the Eclipse IDE. Afterwards, the variant is used to present an outline to the developer. Here, we consider an adapted version of this idea. In detail, we use a full configuration, in which the feature context $\mathcal{FC}$ and a minimized number of additional features are selected. As we use the resulting variant to collect accessible API members for the feature context, we call the approach *minimal variant*.

Similar to our previous strategy, we take a look at our example of product line *BankApplication* with the feature context *Transaction*. To get information about accessible API members using the approach *minimal variant*, we need access to the whole multi product line including the source code of product line *BankAccount*. Based on this assumption, we have to create a configuration of this multi product line in which the feature *Transaction* is selected. Furthermore, we have to select a minimal set of other features so that we get a valid configuration of this multi product line. In detail, we have to select the feature *BankApplication* from the product line *BankApplication*, and the features *SimpleLock*, *Lock* and *BankAccount* from the reused product line *BankAccount*

(cf. Figure 2.14). Based on this full configuration, we create a product-line variant and get an overview of all API members that are included in this variant.

Our previous example shows that this approach can be used to get an overview of accessible API members from a specific feature context. However, it is possible that multiple variants with a minimal set of features exist and, thus, the decision for one or the other variant influences the result of accessible API members. For instance, assuming that the constraint *Transaction* → *Lock* exists instead of the constraint *Transaction* → *SimpleLock* (cf. Figure 2.14). Then, the decision of selecting feature *SimpleLock* instead of *TimeUnitLock* (i.e., the abstract feature without code artifacts) influences the API members that are present in the resulting variant. In detail, because of the selection of feature *SimpleLock*, the method `lock` is included in the resulting set of API members. However, this method only exists in variants where feature *SimpleLock* is also included and, thus, the method is not always safely accessible from our feature context *Transaction*.

Based on our previous example, we can see that the set of features we have to select for the approach *minimal variant* is ambiguous. Therefore, the result of this approach is partly random so that we classify the approach as unsound. Nevertheless, one specific case exists in which the approach can present sound results. In detail, if we assume that we have the information about all variants of a selected feature context, we can compare the members of each variant with all others. If the sets of API members of all these variants are identical, the approach presents a sound result. However, if only one variant presents an API member that is not included in all the others and we randomly pick this variant, the result is unsound. Considering the completeness criteria, we can conclude that the approach is complete as it presents all accessible API members for a specific feature context. In addition, the application of this approach for multi product lines is also possible but with limitations. In detail, we have to keep in mind that multi product lines are in general more complex and, thus, the described problem of soundness is increased. Furthermore, if not all features of the reused product line are of our interest, it is hard to filter the members we are interested in.

5.1.3 Approach 3: Always Available

The approach *always available* is our third approach. By contrast to the previous ones, the approach is not based on the feature context. Instead, we use information of the feature model (i.e., information about the core features, see Definition 3) for the computation of safely accessible members. The approach also comes from our experiences with students' behavior when searching for safely accessible API members (i.e., searching for sound results). In detail, core features are included in all products [Benavides et al. 2010; Trinidad and Ruiz-Cortés 2009] and, thus, also their API members are accessible in all products and all features (cf. *Feature-Model Analyses* of Section 2.1.1 and Definition 3).

Again, we take a look at our example of the product line *BankApplication* with the feature context *Transaction*. For the approach *always available* the feature context is

not relevant, as the computation is based on the core features. Therefore, we neglect the feature context. Using the composed feature model of product line *BankApplication* with product line *BankAccount* (cf. Figure 2.14), we determine the core features of this multi product line. As a result, we only get the feature *BankApplication*. Afterwards, we collect all API members defined in this feature to get an overview of safely accessible API members that can be used in our feature context *Transaction*.

Since the approach is independent of the feature context, we have advantages and drawbacks at the same time. On the one hand, it is not necessary to recompute the set of API members for each feature context and, thus, we can reduce computational effort. Furthermore, as the core features are part of all products, the outcome is sound. On the other hand, the result is not tailored to the feature context so that the approach is not able to present complete results in all cases (i.e., it only presents complete results for core features). In addition to the application to product lines, the approach is also applicable to multi product lines. However, similar to the approach *feature module*, we have some limitations for multi product lines. In detail, it could be that no feature of the reused product line is a core feature. As a result, we get no information about API members defined in the reused product line.

5.1.4 Discussion - How an Optimal Approach Looks Like

As we can conclude from the previous investigation, there is no approach that is complete and sound at the same time. Furthermore, there is no approach that can be used for multi product lines without limitations. Therefore, all these approaches are not suitable to create an overview of safely accessible API members for product lines and multi product lines, as the results hinder the implementation or cause compile-time errors. In Table 5.1, we summarize the results regarding the soundness, completeness, and the ability to apply the approaches to multi product lines.

Focusing on the ability to create a sound and complete overview of accessible API members in product lines, we can summarize the results as follows. The approaches *feature module* and *always available* are sound and, thus, prevent developers from introducing compile-time errors. However, at the same time the approaches are incomplete and hinder an optimal usage of reusable members. In contrast, a *minimal variant* presents all members that can be reused (i.e., it is complete) but at the same time it also presents unsound members which can result in compile-time errors. In contrast to our investigated approaches, a developer is interested in an approach that is sound *and* complete at the same time.

Besides the application of these approaches for product lines, we also consider their application to multi product lines. As a result of our investigations, we can conclude that the approach *feature module* cannot be used to get information of accessible API members from a reused product line. The reason is, that this approach can only present API members from the current feature context. For instance, if we are interested in accessible members for the feature context *Transaction* of product line *BankApplication*, we get no information about members from the product line *BankAccount*. Furthermore,

Approaches	Complete	Sound	Application to Multi Product Lines
Feature Module		✓	
Minimal Variant	✓		(✓)
Always Available		✓	(✓)

Table 5.1: An overview of the classification of state-of-the-art approaches regarding their facility to present safely accessible members when implementing a (multi) product line (checkmarks in brackets illustrate limitations).

the approaches *minimal variant* and *always available* are able to present accessible members from reused product lines but only in certain cases. However, for the approach *minimal variant*, this is only the case if the feature context implies a selection of a feature from the reused product line. In contrast, for the approach *always available*, a feature of the reused product line has to be a core feature in the multi product line. In sum, there is no approach with suitable results that we can use in the context of multi product lines to get safely accessible API members.

Even if the application of the approaches would be suitable for multi product lines, all the approaches use the direct dependencies to a reused product line to get information about accessible API members. Thus, the approaches are not compatible to our concept of multi-level interfaces. Therefore, we propose the concept of a variable interface as syntactical interface, in which we collect all API members of a reused product line with variability information. Based on this variable interface, we also propose the feature-context interface, that is a non-variable view tailored to a feature context to solve the problem of finding safely accessible API members in multi product lines and single product lines.

5.2 The Variable Interface

In the last section, we illustrated that it is a hard task to identify safely accessible API members for a feature context in a product line and in a scenario of a multi product line. In detail, as a developer, we need to know all the dependencies inside of the (multi) product line, even if we are not the domain experts. Therefore, we now introduce the variable interface to ease this task by showing an API with additional variability information for each API member. In the following, we take a look at the definition of a variable interface, present an algorithm to create the variable interface, and we use our running example to demonstrate how to use it. Afterwards, we give an overview of advantages and drawbacks when using the variable interface regarding the implementation of a (multi) product line.

5.2.1 Definition of the Variable Interface

The variable interface is a list of all product line's API members (i.e., classes, methods, and fields) with additional variability information for each member. In detail, each API

member of the variable interface is represented by a unique *signature* with a presence condition that describes under which condition the API member is accessible. As we use FOP for the implementation of our running example, the presence condition is a disjunction of all features, in which the API member is contained in the corresponding feature module. Therefore, we use a simple list to represent the presence condition for API members in FOP. Because of these characteristics, we call the representation a variable interface.

To investigate the variable interface and its facilities to support the developer during the implementation of a (multi) product line, we need to formalize the variable interface $\mathcal{VI}$. We start with a formalization of the set of all API members $\mathcal{AM}$ (i.e., all classes, methods, and fields) of a product line, as it is a base element of $\mathcal{VI}$'s definition. In Definition 27, we define the set of all members $\mathcal{AM}$ as the union of all signatures that we extracted from each feature module of the product line using function *sig*. In detail, function *sig* uses the implementation of a feature module $impl(f_i)$ with $f_i \in \mathcal{F}$ to return a set of all member's signatures of this feature module (cf. Definition 7).

Definition 27. Set of all signatures according to [Schröter et al. 2014]**:**

$$\mathcal{AM} = sig(impl(f_1)) \cup ... \cup sig(impl(f_n))$$

In Definition 28, we use the set of all members $\mathcal{AM}$ to formalize the variable interface $\mathcal{VI}$. In detail, the variable interface is defined as a set of tuples $(m, \mathcal{F}_m)$. Here, m is one specific signature of an API member of the set $\mathcal{AM}$, and $\mathcal{F}_m$ is the set of all features in which the member m is defined.

Definition 28. The variable interface according to [Schröter et al. 2014]**:**

$$\mathcal{VI} = \{(m, \mathcal{F}_m) \mid m \in \mathcal{AM}, \mathcal{F}_m \subseteq \mathcal{F}\}$$

5.2.2 Generating the Variable Interface for FOP

For the generation of the variable interface $\mathcal{VI}$, we provide the corresponding algorithm CreateVariableInterface in Figure 5.1 (cf. Lines 1–9). We create a set, denoted as $\mathcal{VI}$, that represents the variable interface initialized with an empty set (cf. Line 2). Afterwards, we consider the implementation artifacts of each feature f_i (cf. Lines 3–8). In detail, we use the corresponding feature module of feature f_i (i.e., $impl(f_i)$) and determine all contained API members using function *sig* (cf. Line 4). For each member m, we investigate if we have to create a new tuple for $\mathcal{VI}$ or if it is necessary to update the presence condition of an existing tuple by feature f_i. For this purpose, we determine the presence condition of the current member m using algorithm GetFeatures (cf. Line 5, and Lines 10–14). If the presence condition returned by algorithm GetFeatures is an empty set, we know that a new tuple needs to be created. For that reason, we create a new tuple using member m and the empty set $\mathcal{F}_m$ and add this tuple to the variable interface (cf. Line 7). However, in both cases in which the tuple was newly created or only accessed, we subsequently add the current feature f_i to the presence condition $\mathcal{F}_m$ (cf. Line 8). Afterwards, we repeat the procedure for each member of each feature module to determine the complete variable interface $\mathcal{VI}$.

```
1: function CreateVariableInterface(ℱ)
2:     𝒱ℐ := ∅
3:     for f_i ∈ ℱ do
4:         for m ∈ sig(impl(f_i)) do
5:             ℱ_m := GetFeatures(𝒱ℐ, m)
6:             if (ℱ_m = ∅) then
7:                 𝒱ℐ := 𝒱ℐ ∪ {(m, ℱ_m)}
8:             ℱ_m := ℱ_m ∪ {f_i}
9:     return 𝒱ℐ

10: function GetFeatures(𝒱ℐ, m_in)
11:     for (m, ℱ_m) ∈ 𝒱ℐ do
12:         if (GetFirstElement((m, ℱ_m)) = m_in) then
13:             return GetSecondElement((m, ℱ_m))
14:     return ∅
```

Figure 5.1: Algorithm CreateVariableInterface to create the variable interface $\mathcal{VI}$.

Running Example

Using our running example of the product line *BankAccount* that should be reused in the product line *BankApplication*, we illustrate the application of our algorithm. Starting with the feature module of feature *BankAccount* as it is the root feature, we found the class `Account`, as first API member. Since the variable interface $\mathcal{VI}$ is empty, the algorithm GetFeatures also results in an empty set. Therefore, the algorithm creates a new tuple (Line 7) and adds the presence condition with feature *BankAccount* (B) to the set $\mathcal{F}_m$ (Line 8). Similar to the class `Account`, new tuples are created for all API members of the feature module *BankAccount* (e.g., method `update`) because all members have to be initialized. Afterwards, we consider feature module *DailyLimit*. In this case, we also find a definition of the class `Account`. For this member, the algorithm GetFeatures returns a non-empty set of presence conditions in which the feature *BankAccount* is already included. Therefore, we add the feature *DailyLimit* (D) to this set and continue with the next member. If all API members of all feature modules were considered, we get the final variable interface and, thus, we return $\mathcal{VI}$. In Figure 5.2, we present the variable interface to illustrate the outcome of algorithm CreateVariableInterface.

5.3 Filtering According to the Features of Interest

In the previous section, we have shown how to create a variable interface $\mathcal{VI}$ that we want to use as syntactical interface. This variable interface is a complete interface which contains all members of a product line with a corresponding presence condition for each API member. For this reason, the variable interface also contains API members that are only available in hidden features compared to the feature-model interface (i.e., the

$$
\begin{aligned}
&\mathcal{VI}_{BankAccount} = \{ \\
&\quad (class\ Account, \{B, D, I, O, W, E, S\}), \\
&\qquad (Account.calculateInterest() : int, \{I\}), \\
&\qquad (Account.credit(int) : boolean, \{W\}), \\
&\qquad (Account.estimatedInterest(int) : int, \{E\}), \\
&\qquad (Account.getOverdraftLimit() : int, \{B, O\}), \\
&\qquad (Account.isLocked() : boolean, \{S\}), \\
&\qquad (Account.lock() : void, \{S\}), \\
&\qquad (Account.unlock() : void, \{S\}), \\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D\}), \\
&\qquad (Account.update(int) : boolean, \{B, D\}), \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 5.2: Variable interface of product line *BankAccount* (excerpt of class `Account` without fields).

members are available in features that are not of our interest). Thus, the presented variable interface $\mathcal{VI}$ with all API members is in contrast to our initial idea of a syntactical interface as it is not tailored to the features of interest defined by the feature-model interface (cf. Chapter 3). Furthermore, even if we use the variable interface to identify accessible code members, it is still a big effort to identify safely accessible API members that we can use in a specific feature context. Therefore, we now investigate two filtering techniques that can be used to tailor the variable interface to the features of interest that are given by the feature-model interface. In detail, we start with a simple technique with several limitations and present an advanced technique afterwards.

5.3.1 Simple Filtering According to the Features of Interest

In this section, we present a simple filtering technique for the variable interface $\mathcal{VI}$. In detail, it allows us to filter the variable interface to API members that are present if we consider the features of interest from the feature-model interface. We introduce the corresponding algorithm, illustrate the results using our running example, and investigate the advantages and disadvantages of this filtering technique.

The simple filtering technique creates a new and reduced variable interface in which only API members of the features of interest are included (i.e. features of the feature-model interface). In Figure 5.3, we present the algorithm CreateVIntSimple for our simple filtering technique. The algorithm takes the complete variable interface $\mathcal{VI}$ and the features of interest $\mathcal{F}_{Int}$ as input and returns a filtered variable interface. To this end, the algorithm starts to create a new empty set to initialize the simple variable interface $\mathcal{VI}_{Simple}$ (cf. Line 2). Afterwards, we consider for each tuple (cf. Lines 3–6) if the presence condition contains a feature of interest (cf. Line 4). In the positive case,

1: **function** CreateVIntSimple($\mathfrak{VI}$,$\mathcal{F}_{Int}$)
2: $\quad\mathfrak{VI}_{Simple} := \emptyset$ // Init simple variable interface
3: $\quad$**for** $(m, \mathcal{F}_m) \in \mathfrak{VI}$ **do**
4: $\quad\quad$**if** (GetSecondElement($(m, \mathcal{F}_m)$) $\cap \mathcal{F}_{Int}) \neq \emptyset$)) **then**
5: $\quad\quad\quad\mathcal{F}_{Simple} :=$ GetSecondElement($(m, \mathcal{F}_m)$) $\cap \mathcal{F}_{Int}$
6: $\quad\quad\quad\mathfrak{VI}_{Simple} := \mathfrak{VI}_{Simple} \cup$ (GetFirstElement($(m, \mathcal{F}_m)$), $\mathcal{F}_{Simple}$)
7: $\quad$**return** $\mathfrak{VI}_{Simple}$

Figure 5.3: Simple filtering technique for the variable interface that creates a new variable interface $\mathfrak{VI}_{Simple}$ in which only members of the features of interest are included.

we create a new presence condition that only consists of features of interest (cf. Line 5) and create a new tuple as the new entry that we add to our simple variable interface $\mathfrak{VI}_{Simple}$ (cf. Line 6). After the consideration of all tuples, we return the new filtered variable interface (cf. Line 7).

$$
\begin{aligned}
&\mathfrak{VI}_{BankAccount} = \{\\
&\quad(class\ Account, \{B, D, I, S\}),\\
&\quad\quad(Account.calculateInterest() : int, \{I\}),\\
&\quad\quad(Account.getOverdraftLimit() : int, \{B\}),\\
&\quad\quad(Account.isLocked() : boolean, \{S\}),\\
&\quad\quad(Account.lock() : void, \{S\}),\\
&\quad\quad(Account.unlock() : void, \{S\}),\\
&\quad\quad(Account.undoUpdate(int) : boolean, \{B, D\}),\\
&\quad\quad(Account.update(int) : boolean, \{B, D\}),\\
&\quad\quad...\\
&\}
\end{aligned}
$$

Figure 5.4: Simple filtered variable interface of product line *BankAccount*.

Running Example

Using our running example, we apply the simple filtering to our variable interface $\mathfrak{VI}$ of product line *BankAccount*. First, we determine the variable interface $\mathfrak{VI}$ using the algorithm CreateVariableInterface with the set all features of product line *BankAccount* as input to create the variable interface of Figure 5.2. Afterwards, depending on the product line in which the *BankAccount* should be reused, we filter the resulting set using the set of features from the feature-model interface (i.e., the features of interest). In our case, the relevant features are *BankAccount* (B), *DailyLimit* (D), *Interest* (I), and *SimpleLock* (S). Therefore, we use the algorithm CreateVIntSimple with the features of interest and the complete variable interface as input. Then, we consider each member of the interface in a stepwise manner starting with the class

`Account`. This class is defined in at least one of our features of interest (i.e., in this case each feature of interest is part of the presence condition) and, thus, it has also be a part of the reduced variable interface $\mathcal{VI}_{Simple}$. Therefore, we use the existing presence condition of class `Account` $\mathcal{F}_m$ and the set of features of interest $\mathcal{F}_{Int}$ and create the intersection. The resulting set represents the new presence condition for class `Account` of our filtered variable interface. Thus, we create a new entry for the simple variable interface $\mathcal{VI}_{Simple}$ using the class `Account` (i.e., represented as m) and the new presence condition $\mathcal{F}_{Simple}$. Afterwards, we continue with all other entries of the variable interface $\mathcal{VI}$ and create new entries for the simple variable interface $\mathcal{VI}_{Simple}$ if it is necessary. In Figure 5.4, we depict the result of algorithm CreateVIntSimple.

Discussion

The simple filtering technique is an easy-to-use mechanism to filter an existing variable interface to the specific needs of a multi product line. The advantage of the technique is the simplicity and the minimal effort to create it. Furthermore, the technique noticeable reduces the number of API members (i.e., depending on the number of features of interest). This can have a positive effect on the usability as a large number of entries in the interface can be overwhelming for the user. Apart from this positive effect in theory, we also consider the reduced number of entries as a problem. For instance, if the set of features of interest only contains one feature, which is on the lower levels of the tree, e.g., on a leaf node, then we only consider API members of this single feature. However, because of the dependencies in the feature model, we know that all API members of the hidden parent feature are also part of each product in which the feature of interest is included. But in the simple variable interface $\mathcal{VI}_{Simple}$ this information is missing.

5.3.2 Advanced Filtering Using Feature-Model Dependencies

By contrast to the simple filtering technique, the advanced filtering technique also allows us to use API members of hidden features that are accessible if the features of interest are used. All other API members with no dependencies to the features of interest are removed. To this end, the technique also takes indirect dependencies to other features into account using the information of the feature model $\mathcal{M}$ (i.e., the feature model represented as propositional formula). As a result, the number of API members of the variable interface $\mathcal{VI}_{Adv}$ is greater or equal compared to the variable interface $\mathcal{VI}_{Simple}$ and it is less or equal to the number of API members in the complete variable interface $\mathcal{VI}$. In the following, we present the main idea behind this technique, give an algorithm for the interface generation, and exemplify the technique using our running example. Afterwards, we discuss advantages and drawbacks of the advanced filtering technique.

The filtered variable interface created by the advanced filtering technique results in a new variable interface $\mathcal{VI}_{Adv}$ in which all API members of the features of interest and members of other dependent features (i.e., according to the feature model) are included. To this end, we only remove entries of API members that are completely independent from the features of interest (i.e., independent from the features of the feature-model

interface). In detail, we execute two tasks to apply the advanced filtering on a given variable interface $\mathcal{VI}$.

- We extend the presence condition of the API members so that the dependencies to the features of interest are also represented by the member's presence condition. For this reason, we consider each feature of interest as a feature context $\mathcal{FC}$ and are interested in all dependent members from this point of view. As a reminder, we denote the feature context $\mathcal{FC}$ for FOP as one feature in the set of all features $\mathcal{F}$ ($\mathcal{FC} \in \mathcal{F}$) for which we currently implement, maintain or extend the corresponding feature module. Thus, if we consider a feature of interest as a feature context, we assume that we currently work on the corresponding feature module and we search for safely accessible API members for this point of view. Using this information about safely accessible API members for the specific feature of interest, we can update the presence condition of these API members by adding the feature context $\mathcal{FC}$ to the corresponding presence condition. Looking at this intermediate result of the interface, it seems that the specific API is defined in the feature of interest.
- We filter the intermediate result by the simple filtering technique. As the presence condition of each API member was extended, the simple filtering removes equal or less API members compared to a simple filtering without the extended presence condition.

As it is essential to determine the safely accessible API members for a feature context (resp. feature of interest) to apply our advanced filtering technique, we now present a note on how to determine these API members.

An Excursus on Determining Safely Accessible API Members for a Feature Context

The investigation of state-of-the-art approaches in Section 5.1 has shown that no existing approach is able to present safely accessible API members that are sound and complete at the same time. Before we implement our ideas, we need to formally define the set of safely accessible API members $\mathcal{AM}_{Acc}$ for a given feature context.

Definition 29. Safely accessible API members $\mathcal{AM}_{Acc}$ for a feature context $\mathcal{FC}$ according to [Schröter et al. 2014]**:**

$$\mathcal{AM}_{Acc} = \{m \mid (\mathcal{M}_{\mathcal{FC}} \models Constraint_m), (m,\ \mathcal{F}_m) \in \mathcal{VI}\} \tag{29.1}$$

where:

$$\mathcal{M}_{\mathcal{FC}} = \mathcal{M} \wedge \mathcal{FC} \tag{29.2}$$

$$Constraint_m = (\bigvee_{f \in \mathcal{F}_m} f) \mid (m, \mathcal{F}_m) \in \mathcal{VI} \tag{29.3}$$

To define the set of safely accessible members $\mathcal{AM}_{Acc}$, we evaluate the presence condition of each API member m according to the feature model $\mathcal{M}$ and the feature context $\mathcal{FC}$. For this purpose, we use propositional formulas to represent all the components. In Definition 29, we present the complete formalization of the set $\mathcal{AM}_{Acc}$ with secondary conditions. To ease the comprehension, we start to investigate the secondary conditions. First, we need a propositional formula that describes all valid configurations in which the feature context $\mathcal{FC}$ is included (i.e., a propositional formula of the feature model with the partial configuration $(\{\mathcal{FC}\}, \emptyset)$, see also Section 2.1.1). For this purpose, we use the conjunction of the feature model's propositional formula $\mathcal{M}$ and the propositional variable that corresponds to the feature context to create the propositional formula $\mathcal{M}_{FC}$ (cf. Equation 29.2). Second, we need to transform the presence condition of our representation $\mathcal{F}_m$ as propositional formula $Constraint_m$. As mentioned earlier, for product lines written with FOP, this is the disjunction of all features in which the specific member is defined (cf. Equation 29.3). Based on these secondary conditions, we can define the set $\mathcal{AM}_{Acc}$ of safely accessible API members for the feature context $\mathcal{FC}$ (cf. Equation 29.1). In detail, we check for each tuple in $(m,\ \mathcal{F}_m)$ of $\mathcal{VI}$ whether the implication of the partial configuration $\mathcal{M}_{\mathcal{FC}}$ and the presence condition is a tautology $(\mathcal{M}_{\mathcal{FC}} \rightarrow Constraint_m)$. In other words, we investigate if the propositional formula $\mathcal{M}_{\mathcal{FC}}$ ensures that at least one feature of the presence condition is included in each valid variant of the partial configuration. If this is the case, the API member is safely accessible from feature context $\mathcal{FC}$ and, thus, we add this feature to the set $\mathcal{AM}_{Acc}$.

Algorithm for the Advanced Filtering of the Variable Interface

For the application of the advanced filtering technique, we create a new algorithm CreateVIntAdvanced and depict the corresponding pseudo code in Figure 5.5. The algorithm uses the variable interface $\mathcal{VI}$, the features of interest $\mathcal{F}_{Int}$, and the feature model $\mathcal{M}$ as input and returns the new advanced variable interface $\mathcal{VI}_{Adv}$. First, we create a copy of the variable interface $\mathcal{VI}$ and initialize an advanced variable interface $\mathcal{VI}_{Adv}$ with this copy (cf. Line 2). Afterwards, we consider all features of interest, i.e., all features of our feature-model interface (cf. Lines 3–8). In detail, we consider each feature of interest as feature context and, thus, we create the corresponding set of accessible API members using algorithm GetAccessibleMembersFromFeature (cf. Line 4). Afterwards, we check for each entry in the advanced variable interface $\mathcal{VI}_{Adv}$ if the corresponding member m is also part of the resulting set of accessible members (cf. Lines 5–8). If the member is also part of the current set of accessible members (cf. Line 6), we know that from the point of view of the feature context $\mathcal{FC}$ (i.e., f_{Int}), the member is also accessible. Therefore, we can add the current feature context to the presence condition (cf. Line 8). After updating all presence conditions of all entries in the advanced variable interface $\mathcal{VI}_{Adv}$, we can use the algorithm CreateVIntSimple to remove all entries in which the presence condition does not contain a feature of interest (cf. Line 9) and return the new variable interface $\mathcal{VI}_{Adv}$.

In the second part of Figure 5.5, we present algorithm GetAccessibleMembers-FromFeature (cf. Lines 10–18) that is used in our previous description for the ad-

vanced filtering of the variable interface $\mathcal{VI}_{Adv}$. The algorithm uses the variable interface $\mathcal{VI}$, a feature of interest f_{Int}, and the feature model $\mathcal{M}$ as input to determine the safely accessible members from the viewpoint of feature f_{Int}. As first step, we initialize the result variable $\mathcal{AM}$, with an empty set (cf. Line 11). In detail, $\mathcal{AM}$ is used to store the accessible API members. Afterward, we create a propositional formula that represents the partial configuration of the feature model $\mathcal{M}$ with feature f_{Int} (cf. Line 12). Using the propositional formula of the partial configuration, we consider each tuple of the variable interface $\mathcal{VI}$ step by step and check whether the API member is accessible from the feature f_{Int} (cf. Lines 13–17). In detail, we create the propositional formula of the presence condition $Constraint_m$ for the API member m (cf. Line 14). Then, we check whether the implication of the partial configuration $\mathcal{M}_f$ and the presence condition of the API member m is a tautology. For this reason, we create a propositional formula $\mathcal{M}_m$ for the API member m that represents this implication (cf. Line 15). If $\mathcal{M}_m$ is a tautology, we know that the member is always accessible from feature f_{Int} (cf. Line 16). Consequently, the member is added to the set of accessible members $\mathcal{AM}$ (cf. Line 17). After considering all tuples of the variable interface $\mathcal{VI}$, we finally return the set of safely accessible members $\mathcal{AM}$ (cf. Line 18).

```
1:  function CreateVIntAdvanced(VI, F_Int, M)
2:      VI_Adv := VI
3:      for f_Int ∈ F_Int do
4:          AM_Acc := GetAccessibleMembersFromFeature(VI, f_Int, M)
5:          for (m, F_m) ∈ VI_Adv do
6:              if (GetFirstElement((m, F_m)) ∈ AM_Acc) then
7:                  F_m := GetSecondElement((m, F_m))
8:                  F_m := F_m ∪ {f_Int}
9:      return CreateVIntSimple(VI_Adv, F_Int)

10: function GetAccessibleMembersFromFeature(VI, f_Int, M)
11:     AM := ∅
12:     M_f := M ∧ f_Int
13:     for (m, F_m) ∈ VI do
14:         Constraint_m := (⋁_{f ∈ GetSecondElement((m, F_m))} f)
15:         M_m := M_f → Constraint_m
16:         if (!isSatisfiable(¬M_m)) then
17:             AM := AM ∪ {m}
18:     return AM
```

Figure 5.5: Advanced filtering technique for the variable interface that creates a new variable interface $\mathcal{VI}_{Adv}$ in which members of the features of interest and members of dependent features are included.

Running Example

We illustrate our algorithm using our running example of product line *BankAccount*. In this case, we assume that the variable interface $\mathfrak{VI}$ of the complete product line already exists (cf. Figure 5.2). Furthermore, we plan to use product line *BankAccount* in the multi product line *BankApplication* and, thus, we know that the features *BankAccount* (B), *DailyLimit* (D), *Interest* (I) and *SimpleLock* (S) are the features of interest. Based on this information and the feature model $\mathcal{M}$ of the product line *BankAccount*, we create the corresponding advanced variable interface $\mathfrak{VI}_{Adv}$. In detail, we start to create a copy of the input variable interface $\mathfrak{VI}$. Afterwards, we take a look at all features of interest. We start with feature *BankAccount* and assume that this is the current feature context $\mathcal{FC}$. In detail, we determine all accessible members from the viewpoint of feature *BankAccount* using algorithm GetAccessibleMembersFromFeature (cf. Figure 5.5, Line 4). Afterwards, we update the presence condition of all resulting members (e.g., method `update`) in the variable interface $\mathfrak{VI}_{Adv}$ so that also the current feature context *BankAccount* is included in the presence condition (cf. Lines 5–8). In the case of feature *BankAccount*, the presence condition of all accessible members already contains the feature *BankAccount* (B) and, thus, the presence condition does not change (cf. Figure 5.2). As next feature of interest, we consider feature *SimpleLock*. Again, we determine all accessible API members from this point of view. As the result, we also get the safely accessible API members `update`, `undoUpdate`, and `getOverdraftLimit`. However, the original presence condition of these API members does not contain feature *SimpleLock* (S) (cf. Figure 5.2). To this end, we update the corresponding presence conditions by feature *SimpleLock* and continue with the next feature of interest. In Figure 5.6, we present the intermediate result of the advanced filtering. Now, we can use the simple filtering to remove unnecessary API members (cf. Line 18). The final outcome of the advanced filtering is depicted in Figure 5.7.

In the previous description, we also used the algorithm GetAccessibleMembersFromFeature to get safely accessible API members from the viewpoint of a given feature. The algorithm is an essential part of our algorithm. Therefore, we also exemplify the algorithm using our running example of product line *BankAccount*. In detail, we use the variable interface $\mathfrak{VI}$ (cf. Figure 5.2), the feature *DailyLimit* (D), and the feature model (cf. Figure 2.1) represented as propositional formula $\mathcal{M}_{BankAccount}$ of product line *BankAccount* as input. As the result of this input, we want to determine the safely accessible API members from the viewpoint of feature *DailyLimit*. As starting point, we create the set $\mathcal{AM}$ to store the corresponding API members. Afterwards, we create a new propositional formula $\mathcal{M}_f$ that represents a partial configuration of $\mathcal{M}_{BankAccount}$ in which the feature *DailyLimit* (D) is selected (cf. Figure 5.5, Line 15). To determine accessible API members, we then consider all tuples of the variable interface $\mathfrak{VI}$ (cf. Figure 5.2). For the purpose of illustration, we now consider the tuple $(Account.update(int) : boolean, \{B, D\})$. We transform the presence condition $\{B, D\}$ to the propositional formula $Constraint_m$ using a disjunction of all contained features (i.e., $B \vee D$). Then, we create a new propositional formula $\mathcal{M}_m$ to check whether the partial configuration $\mathcal{M}_f$ always ensures that one of the features *BankAccount* (B), or

$$
\begin{aligned}
&\mathcal{VI}_{BankAccount} = \{\\
&\quad (class\ Account, \{B, D, I, O, W, E, S\}),\\
&\qquad (Account.calculateInterest() : int, \{I\}),\\
&\qquad (Account.credit(int) : boolean, \{W\}),\\
&\qquad (Account.estimatedInterest(int) : int, \{E\}),\\
&\qquad (Account.getOverdraftLimit() : int, \{B, D, I, O, S\}),\\
&\qquad (Account.isLocked() : boolean, \{S\}),\\
&\qquad (Account.lock() : void, \{S\}),\\
&\qquad (Account.unlock() : void, \{S\}),\\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D, I, S\}),\\
&\qquad (Account.update(int) : boolean, \{B, D, I, S\}),\\
&\qquad ...\\
&\}
\end{aligned}
$$

Figure 5.6: Intermediate result (without a final filtering with the simple filtering technique) for the computation of the variable interface $\mathcal{VI}_{Adv}$ using the advanced filtering technique according to product line *BankAccount* with our features of interest.

$$
\begin{aligned}
&\mathcal{VI}_{BankAccount} = \{\\
&\quad (class\ Account, \{B, D, I, S\}),\\
&\qquad (Account.calculateInterest() : int, \{I\}),\\
&\qquad (Account.getOverdraftLimit() : int, \{B, D, I, S\}),\\
&\qquad (Account.isLocked() : boolean, \{S\}),\\
&\qquad (Account.lock() : void, \{S\}),\\
&\qquad (Account.unlock() : void, \{S\}),\\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D, I, S\}),\\
&\qquad (Account.update(int) : boolean, \{B, D, I, S\}),\\
&\qquad ...\\
&\}
\end{aligned}
$$

Figure 5.7: Final result of the variable interface $\mathcal{VI}_{Adv}$ using the advanced filtering technique according to product line *BankAccount* with our features of interest.

DailyLimit (D) also exists in each resulting variant. In the case of method `update`, the implication is a tautology and, thus, we know that the method is safely accessible from feature *BankAccount*. Therefore, we add this method to the set $\mathcal{AM}$ and continue with the next tuple. If all tuples were considered, the algorithm returns the final set $\mathcal{AM}$ that we depict in Figure 5.8.

$$
\begin{aligned}
&\mathcal{AM}_{\mathcal{FC}=BankAccount} = \{ \\
&\quad class\ Account, \\
&\qquad getOverdraftLimit() : int, \\
&\qquad Account.undoUpdate(int) : boolean, \\
&\qquad Account.update(int) : boolean, \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 5.8: Result of the computation of all accessible members using algorithm GetAccessibleMembersFromFeature with the variable interface of product line *BankAccount* $\mathcal{VI}_{BankAccount}$, feature *DailyLimit* and the corresponding feature model $\mathcal{M}$ as input.

Discussion

The advanced filtering technique is an automatic technique to filter the variable interface to all API members that are accessible if the features of interest are used. This also includes API members that are not directly defined in the features of interest and, thus, it contains members of hidden features. Like the simple technique that we described in the previous section, the advanced technique does not induce manual effort for the developer to create a tailored variable interface. By contrast, the advanced filtering technique only neglects API members from the complete variable interface that are completely optional. As the result, the number of API members in the variable interface $\mathcal{VI}_{Adv}$ is equal or bigger compared to the variable interface $\mathcal{VI}_{Simple}$. Indeed the increased number of API members in the variable interface makes the interface less comprehensible. However, compared to the variable interface $\mathcal{VI}_{Simple}$, in which only members of the features of interest are included, the variable interface $\mathcal{VI}_{Adv}$ presents all accessible API members of the underlying product line and, thus, also API members that are not directly included in the features of interest. For instance, if we assume that the feature *SimpleLock* is the only feature of interest, the member `update` is not accessible with the simple technique even though it is essential for a bank account. By contrast, the advanced technique also presents the method `update` because it can be used if the feature *SimpleLock* is used.

5.4 Feature-Context Interfaces

In the previous section, we introduced the variable interface $\mathcal{VI}$ as a basic concept for a syntactical interface according to our multi-level interfaces. Using different filtering techniques, we also discussed the application of the variable interface to ease the implementation of the product line. However, even if each filtered variant of the variable interface is able to support the developer during the implementation step of the multi product line, there is still a remaining manual effort to find API members for a specific

implementation task. Thus, it is still necessary for a developer to understand the dependencies from the multi product line to this interface and to conclude from existing presence conditions whether an API member is accessible. Therefore, we now introduce the concept of feature-context interfaces to overcome this limitation. Afterwards, we present application scenarios in a single and a multi product line.

A feature-context interface is a non-variable view on the variable interface to get an overview of safely accessible API members for a given feature context $\mathcal{FC}$. As a result, the feature-context interface is a plain set of API members that does not contain any variability information. For instance, if we maintain, extend, or implement feature *SimpleLock* of product line *BankAccount*, *SimpleLock* is the feature context and we get the information that method `update` and other API members are accessible in this situation. However, we already described the idea of a feature-context interface and we also described how to create it. In detail, we used the idea of feature-context interfaces to introduce the details of the advanced filtering technique for the variable interface and presented a corresponding note as explanation. In Definition 29, we described the idea of the feature-context interface and we also introduced the algorithm to create a feature-context interface for a feature context. We used it to create our advanced variable interface $\mathcal{VI}_{Adv}$. This means, we can use the algorithm GetAccessibleMembersFromFeature (see Figure 5.5) and apply it to the feature context $\mathcal{FC}$. In the following, we explain the application of feature-context interfaces for a single and a multi product line.

5.4.1 Feature-Context Interfaces of a Single Product Line

Before we take a look at the main application of feature-context interfaces inside of a multi product line, we give an overview of an application for a single product line. As we stated above, the feature-context interface can be considered as a filter on the variable interface $\mathcal{VI}$. So far we only introduced the variable interface $\mathcal{VI}$ in the context of a multi product line in which we want to get an overview of all reusable API members. Nevertheless, the variable interface $\mathcal{VI}$ can also be used for single product lines. As a consequence, the concept of the variable interface and the feature-context interface is also applicable to a single product line to achieve an overview of safely accessible API members for a given feature context.

Running Example

To illustrate the application, we use our running example and consider the product line *BankAccount* as a standalone product line without dependencies to other product lines. Furthermore, we assume that we plan to maintain, extend, or implement feature *InterestEstimation*. To get an overview of safely accessible API members from the viewpoint of feature *InterestEstimation*, we have to execute two steps. First, we create the unfiltered variable interface $\mathcal{VI}$ using algorithm CreateVariableInterface (see Figure 5.1) with the set of all features of product line *BankAccount* as input. Second, we create the feature-context interface $\mathcal{FCI}_{InterestEstimation}$. Therefore, we use algorithm

GetAccessibleMembersFromFeature (see Figure 5.5) with the variable interface $\mathcal{VI}$, the feature *InterestEstimation* as the feature context, and the feature model $\mathcal{M}_{BankAccount}$ as input. As a result, we get an overview of all safely accessible API members that can be used to implement feature *InterestEstimation*. For instance, the class `Account` is included in the feature-context interface as it is a member of all feature modules. In addition, the method `estimatedInterest` is defined in the feature module *InterestEstimation* and, thus, it is also part of corresponding feature-context interface. However, we also get the information that, besides others, the method `update` as part of the core feature *BankAccount* and `calculateInterest` defined in the parent feature *Interest* are accessible from feature context *InterestEstimation*.

5.4.2 Feature-Context Interfaces of a Reused Product Line

In the previous section, we have seen how to benefit from the feature-context interface in single product lines. Now we present insights into the application of feature-context interfaces in the context of multi product lines. Here, we investigate how to create a feature-context interface that presents the accessible API members of a reused product line.

In our multi product line scenario, a product line PL_D (i.e., the dependent product line) reuses a second product line PL_R (i.e., the reused product line) and is interested in the safely accessible members of product line PL_R. We assume that a variable interface $\mathcal{VI}$ (or a respective variable interface $\mathcal{VI}_{Simple}$ or $\mathcal{VI}_{Adv}$) of the product line PL_R already exists and that we have no information about the implementation details behind this variable interface. Furthermore, we assume that we currently maintain, extend, or implement a feature f of product line PL_D in which we plan to reuse members of product line PL_R (i.e., members of the corresponding variable interface $\mathcal{VI}$). Consequently, feature f represents the feature context for which we have to determine the feature-context interface. To calculate the feature-context interface of f, we can also use the algorithm GetAccessibleMembersFromFeature (see Figure 5.5). But in comparison to our previous applications in single product lines, we have to use different parameters. Of course, we use the variable interface $\mathcal{VI}$ of product line PL_R as input, but the feature context is from the product line PL_D and also the feature model $\mathcal{M}$ is different. In detail, $\mathcal{M}$ needs to be the composed feature model based on product line PL_D and the feature-model interface of product line PL_R (i.e., $\mathcal{M}_{PL_D/PL_R}$, see Chapter 4 for more details). Using these parameters, the algorithm GetAccessibleMembersFromFeature determines all accessible members from product line PL_R that we can use in the feature context f.

Running Example

Using our running example, we demonstrate the application of feature-context interfaces for reused product lines. We use feature *Transaction* as the feature context from product line *BankApplication* and we are interested in all API members that we can use from the interface of product line *BankAccount*. Consequently, we use the already

existing $\mathcal{VI}_{Adv}$ of product line *BankAccount* (cf. Figure 5.7), the feature *Transaction*, and a composed feature model as input to apply algorithm GetAccessibleMembersFromFeature. In detail, the used feature model is a composition based on feature model *BankApplication* with the feature-model interface of product line *BankAccount* (i.e., $\mathcal{M}_{BankApplication/Int}$). In Figure 5.9, we present the result of this computation. Besides the API members of the core feature *BankAccount*, the result also presents API members of the feature *SimpleLock*. The reason for this result is the inter-model constraint *Transaction* $\rightarrow$ *SimpleLock* so that all the API members of feature *SimpleLock* are also part of the feature-context interface.

$$
\begin{aligned}
&\mathcal{FCI}_{Transaction} = \{ \\
&\quad class\ Account, \\
&\qquad Account.getOverdraftLimit : int, \\
&\qquad Account.isLocked() : boolean, \\
&\qquad Account.lock() : void, \\
&\qquad Account.unlock() : void, \\
&\qquad Account.undoUpdate(int) : boolean, \\
&\qquad Account.update(int) : boolean, \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 5.9: Feature-context interface of product line *BankAccount* for feature *Transaction* of product line *BankApplication* (excerpt of class `Account` without fields based on the advanced variable interface $\mathcal{VI}_{BankAccount}$).

5.4.3 Complete Feature-Context Interfaces of a MPL

Similar to feature-context interfaces for single product lines, developers of multi product lines also want to get a complete overview of all safely accessible API members. By contrast, the previous application of feature-context interfaces for multi product lines only considers API members of the reused product line. To this end, we now take a look at a second application scenario of feature-context interfaces for multi product lines.

As mentioned above, if we maintain a product line PL_D, we want to know all accessible API members of a reused product line PL_R but also all API members of the product line PL_D. For this purpose it is not enough to create a variable interface $\mathcal{VI}$ of the product line PL_R, we also need a variable interface $\mathcal{VI}_D$ of the product line PL_D. Thus, we use the algorithm CreateVariableInterface (see Figure 5.1) with a set of features, which represents all features defined in product line PL_D, as input to achieve the variable interface $\mathcal{VI}_D$. Afterwards, we can combine the variable interfaces $\mathcal{VI}_D$ and $\mathcal{VI}_R$. If we assume that each API member is represented by the full-qualified name and that the namespaces of both product lines are disjoint, we can use a union to combine both variable interfaces. As the result, we have one variable interface $\mathcal{VI}_{PL_D/PL_R}$ that

represents all API members of the product line PL_D and of the product line PL_R. Thus, we can use algorithm GetAccessibleMembersFromFeature (see Figure 5.5) to create the feature-context interface for the feature context $\mathcal{FC}=f$. In detail, we again use the composed feature model of product line PL_D with the feature-model interface of product line PL_R, the feature f, and the combined variable interface $\mathcal{VI}_{PL_D/PL_R}$ as input. The result is a feature-context interface with all accessible API members of the product line PL_D and product line PL_R for the feature context f.

$$
\begin{aligned}
&\mathcal{VI}_{BankApplication/BankAccount} = \{\\
&\quad (class\ Account, \{B, D, I, S\}),\\
&\qquad (Account.calculateInterest() : int, \{I\}),\\
&\qquad (Account.getOverdraftLimit() : int, \{B, D, I, S\}),\\
&\qquad (Account.isLocked() : boolean, \{S\}),\\
&\qquad (Account.lock() : void, \{S\}),\\
&\qquad (Account.unlock() : void, \{S\}),\\
&\qquad (Account.undoUpdate(int) : boolean, \{B, D, I, S\}),\\
&\qquad (Account.update(int) : boolean, \{B, D, I, S\}),\\
&\quad (class\ Transaction, \{T\}),\\
&\qquad (Transaction.transfer(Account, Account, int) : boolean, \{T\}),\\
&\qquad (Transaction.lock(Account, Account) : boolean, \{T\}),\\
&\qquad ...\\
&\}
\end{aligned}
$$

Figure 5.10: Excerpt of the combined variable interface $\mathcal{VI}_{BankApplication/BankAccount}$.

Running Example

To illustrate how to achieve an overview of safely accessible API members of all parts of a multi product line, we use our running example. In detail, we currently maintain feature *Transaction*. Therefore, this feature represents the feature context and we need an overview of all API members that we can use from product line *BankApplication* and product line *BankAccount*. Assuming an existing variable interface $\mathcal{VI}_{BankAccount}$ of product line *BankAccount* (i.e., based on the simple or advanced filtering), we only have to determine the variable interface of product line *BankApplication*. For this purpose, we use algorithm CreateVariableInterface (see Figure 5.1) with a set of the two concrete features, feature *BankApplication* and *Transaction*. As the result, we get a variable interface $\mathcal{VI}_{BankApplication}$ in which only the API members of these two features are included. Furthermore, as we know that the full-qualified name of all API members is disjoint, we can combine the variable interfaces $\mathcal{VI}_{BankApplication}$ and $\mathcal{VI}_{BankAccount}$ by a union. In Figure 5.10, we depict an excerpt of the combined variable interface $\mathcal{VI}_{BankApplication/BankAccount}$ that we use to create the feature-context interface for the feature *Transaction*. Afterwards, we use algorithm GetAccessibleMembersFromFeature with the variable interface $\mathcal{VI}_{BankApplication/BankAccount}$, the feature *Transaction*,

and the composed feature model of product line *BankApplication* and the feature-model interface as input (i.e., $\mathcal{M}_{BankApplication/Int}$). We achieve the feature-context interface for the feature context *Transaction*, in which all accessible API members of both product lines are included. We depict an excerpt of the result in Figure 5.11. In addition to our example of the previous section (cf. Figure 5.9), several additional API members are included in this feature-context interface. For instance, the method `transfer` is an API member of the product line *BankApplication* and, thus, the method is only part of the feature-context interface if we determine the interface based on the complete variable interface $\mathcal{VI}_{BankApplication/BankAccount}$.

$$
\begin{aligned}
&\mathcal{FCI}_{Transaction} = \{ \\
&\quad class\ Account, \\
&\qquad Account.getOverdraftLimit : int, \\
&\qquad Account.isLocked() : boolean, \\
&\qquad Account.lock() : void, \\
&\qquad Account.unlock() : void, \\
&\qquad Account.undoUpdate(int) : boolean, \\
&\qquad Account.update(int) : boolean, \\
&\quad class\ Transaction, \\
&\qquad Transaction.transfer(Account, Account, int) : boolean, \\
&\qquad Transaction.lock(Account, Account) : boolean, \\
&\qquad ... \\
&\}
\end{aligned}
$$

Figure 5.11: Excerpt of the feature-context interface for feature *Transaction* of multi product line *BankApplication* using the combined variable interface $\mathcal{VI}_{BankApplication/BankAccount}$.

5.5 Evaluation: The Feature-Context Interface in Practice

In this section, we investigate Hypothesis 2 and want to find out whether the variable interface, as representative of the syntactical interface, can help the developer to detect reusable implementation artifacts. For this purpose, we examine the potential of feature-context interfaces that are based on the variable interface to detect safely accessible API members of a product line. In detail, to investigate the potential of feature-context interfaces for the development of product lines as well as multi product lines, we performed a quantitative evaluation. As feature-context interfaces at the same time support the development of product lines as well as multi product lines, it is possible to evaluate both of them. However, in Section 5.4, we illustrated that the application of feature-context interfaces is similar in both scenarios. Therefore and

because of the fact that more single product lines are freely accessible and well-known from other evaluations, we decided to investigate only single product lines.

For the quantitative evaluation, we are interested in showing that feature-context interfaces are useful in supporting the product-line development. For this purpose, we investigate whether state-of-the-art techniques also present unsound (i.e., *minimal variant*) or incomplete (i.e., *always available*, *feature module*) results in practice, so that we can underline the necessity of feature-context interfaces. Furthermore, we want to know in which scenarios feature-context interfaces achieve their full potential compared to the state-of-the-art approaches. To this end, we investigate two questions:

(a) Do state-of-the-art approaches present incomplete or unsound results for existing product lines?
(b) If existing product lines present incomplete and unsound results, how evident are the errors for the developer?

In the following, we explain our experiment design. Afterwards, we present the experiment results and the threats to validity.

5.5.1 Experiment Design

In the following, we present details to our experiment design. In detail, we introduce our subject systems, present details to our experiment and implementation.

Subject Systems

Our study is based on publicly available subjects (cf. Table 5.2). We selected these subjects because of multiple reasons, (a) their implementation language and paradigm, (b) their differences in structure and size, (c) their syntactical correctness, and (d) since they are commonly used for evaluations. To be more specific:

(a) Our concept of feature-context interfaces was designed for product lines written in Java with the paradigm of FOP (see Section 2.1.2 for more details). Furthermore, our implementation is based on the tool FeatureHouse (see next subsection for more details). As a result, all systems are based on these concepts.

(b) The investigated subjects have different sizes regarding the number of features and products. Furthermore, to ensure the domain-independent results, the subjects are also from different domains.

(c) To ensure the correct generation of the results for each state-of-the-art approach and the feature-context interfaces, we only used subjects without compile-time errors.

(d) We only used subjects that were used in previous case studies by other researches [Apel et al. 2013b; Kolesnikov et al. 2013; Thüm et al. 2011a]. As a result, many readers will be familiar with these subjects.

Product Line	Features (Alternative Features)		Products	Unique Classes
DesktopSearcher	22	(8)	462	21
GameOfLife	23	(2)	65	21
GPL	38	(15)	156	16
GraphLib	6	(0)	16	5
Notepad	15	(4)	512	8
PKJab	12	(0)	48	51
TankWar_PC	37	(7)	87 360	21
ZipMe	17	(0)	24	31

Table 5.2: Overview of all subjects for the quantitative evaluation of feature-context interfaces.

In Table 5.2, we give an overview of the subjects we selected based on these criteria. In detail, we investigate the search engine *DesktopSearcher*, two games (*GameOfLife* and *TankWar_PC*), two graph libraries (*GPL* and *GraphLib*), a simple text editor (*Notepad*), the chat client *PKJab*, and the compression library *ZipMe*.

Experiment

To answer our questions, we investigate the existing state-of-the-art approaches and our concept of feature-context interfaces when considering a typical implementation scenario of a developer. In detail, we assume that a developer wants to implement, maintain, or extend a specific feature, such as the feature *InterestEstimation* of our running example. Based on this assumption, we determine the incomplete and unsound results (question (a)) and the frequency of potential errors (question (b)) and compare the respective results. In the following, we present the detailed experiment procedure.

Question (a): Incomplete and Unsound Results. To investigate question (a), we consider each feature step by step as feature context. For each feature context, we determine all API members that each state-of-the-art approach presents to the developer. In detail, we determine the accessible API members given by the approaches *feature module*, *minimal variant*, *always available*, and our feature-context interface. For the purpose of the visualization and to ease the comparison, we also determine the number of API members of the variable interface. We use this value to scale the number of accessible API members given by each state-of-the-art approach.

Whereas the approaches *feature module*, *always available*, and the feature-context interface generate one specific result for each feature context, the results of the approach *minimal variant* depend on the concrete variant, that is used for the generation. However, if multiple minimal variants exist for a feature context, we determine all these variants and determine the set of accessible members. For a fair comparison with the other approaches, for each feature context, we chose the variant with the minimum number of members.

Question (b): Potential Errors. Similar to the previous question, for each approach, we also determine accessible API members so that we can count the resulting potential errors for question (b). Whereas the first question only considers the number of API members that are represented by each approach, this research question counts errors based on a comparison of the signatures. In detail, we investigate each API member of the respective state-of-the-art approach and search for the corresponding member in the feature-context interface. Based on the knowledge that the feature-context interface contains all safely accessible API members, not more or less, this comparison helps to detect potential errors caused by the state-of-the-art approaches. On the one hand, if an API member exists in the state-of-the-art approach but not in the feature-context interface, we count it as an error that can lead to compile-time errors (i.e., the result is unsound). This will happen for the approach *minimal variant* (cf. Section 5.1). On the other hand, if an API member exists in the feature-context interface but not in the state-of-the-art approach, we count it as an error that reduces the reuse potential of the product line (i.e., the result is incomplete). As we know from Section 5.1, the approaches *always available* and *feature module* present these incomplete results. Again, to ease the comparison, we scale all results to the number of API members represented in the variable interface as it is the total number of API members of the product line. Thus, this number of API members represents the theoretical number of potential errors.

Similar to what we discussed for question (a), the exceptional case that multiple minimal variants can exist forces us to modify the way in which we count errors for the approach *minimal variant*. Specifically, we consider all minimal variants of a feature context to detect the number of potential errors.

Implementation Details

To evaluate the concept of feature-context interfaces and to provide the approach to the community, we implemented feature-context interfaces in FeatureIDE [Meinicke et al. 2017; Thüm et al. 2014b]. In the following, we present some details of this implementation.

As FeatureIDE provides the necessary tool support and the most examples for product lines written in Java with FeatureHouse, we also implemented feature-context interfaces for FeatureHouse product lines. In detail, FeatureHouse is a composer for FOP that can be used for multiple languages [Apel et al. 2009, 2013b]. The composition mechanism of FeatureHouse relies on grammar rules, so that only one additional keyword is needed to implement FOP product lines in Java. In detail, using the keyword `original`, we can call an already existing implementation of this method from another feature module. To collect the API members of FeatureHouse product lines, we use a second tool, called FUJI [Apel et al. 2012; Kolesnikov et al. 2013]. FUJI is a compiler and type checker and allows us to perform a type safe comparison of the API members of each feature module. Based on the information given by FUJI, we create a variable interface and store it inside of FeatureIDE. As FeatureIDE is an IDE that provides advanced tool

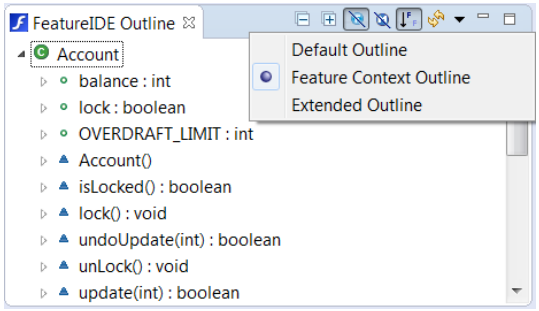


Figure 5.12: Feature-context outline for class `Account` of product line *BankAccount* and feature *SimpleLock*.

Feature Context	Feature-Context Interface	*Minimal Variant*	*Feature Module*	*Always Available*
BasicGraph	12 (0.41)	12 (0.41)	12 (0.41)	12 (0.41)
Color	17 (0.59)	17 (0.59)	8 (0.28)	12 (0.41)
PrintHeader	15 (0.52)	15 (0.52)	5 (0.17)	12 (0.41)
Recursive	15 (0.52)	15 (0.52)	5 (0.17)	12 (0.41)
Weight	18 (0.62)	18 (0.62)	8 (0.28)	12 (0.41)

Table 5.3: Number of accessible members for each state-of-the-art approach considering product line *GraphLib*. The numbers in brackets are the scaled values with respect to the variable interface $\mathfrak{VI}$ with 29 members that we use for the purpose of illustration represented in Figure 5.13.

support for the implementation of product lines [Meinicke et al. 2017; Thüm et al. 2014b], the feature context is automatically given by the file that is currently opened in the editor of the Eclipse IDE. Using this information and the stored variable interface, FeatureIDE generates the corresponding feature-context interface on the fly, so that the information is available in the IDE's outline (cf. Figure 5.12) or as content assist. In contrast, for our evaluation, we assigned our desired feature context automatically and stored the corresponding feature-context interface in FeatureIDE.

To enable a fair and correct comparison of our feature-context interface to the state-of-the-art approaches, we also used Fuji to collect the API members for the approaches *feature module*, *always available*, and *minimal variant*.

5.5.2 Experiment Results and Discussion

First, we present the accessible members for each approach as result for question (a). Second, we present the results regarding question (b) and give an insight into the potential errors of each state-of-the-art approach.

Question (a): Incomplete and Unsound Results. In Figure 5.13, we present the

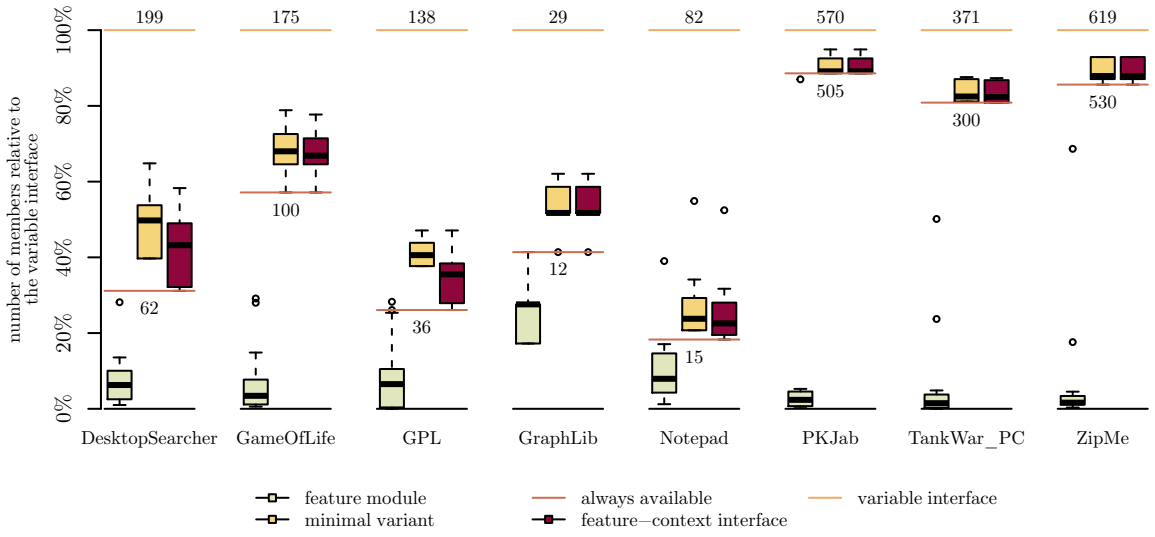


Figure 5.13: Overview of the amount of accessible members for the state-of-the-art approaches and the feature-context interface relative to the variable interface.

results of our investigation regarding the incompleteness and unsoundness of state-of-the-art approaches compared to feature-context interfaces. To ease the comprehension of the graphical representation, we first want to take a look at the data behind this representation. For this purpose, we use the product line *GraphLib*, as it is the smallest product line of our investigation. We present the corresponding data in Table 5.3. In detail, we consider each concrete feature of the product line as the feature context and collect the corresponding number of API members for the feature-context interface, the *minimal variant*, and the *feature module*. Furthermore, even though it is independent of the feature context, we also consider the API members for the approach *always available*. To ease the representation, we use the number of API members in the variable interface $\mathcal{VI}$ (in this case 29 members) to scale all collected values. We use boxplots to illustrate the distribution of the results regarding the approaches feature-context interface, *minimal variant*, and *feature module* (cf. Figure 5.13). To illustrate the distribution of the different values, we use boxplots[5] for the representation of the approaches feature-context interface, *minimal variant*, and *feature module* (cf. Figure 5.13). As the number of members of the variable interface and the approach *always available* are constant, we use a horizontal line for the purpose of representation.

We now present the plain results shown in Figure 5.13 that we will discuss in the next section. The variable interface of the product line *GraphLib* consists of 29 API members (orange line) but the approach *always available* only presents 41 % of these API members (red line). Furthermore, the number of API members that are accessible

[5]As we used R (`http://www.r-project.org/`) for the representation of our results, we also used the default settings for boxplots. In detail, the black line of each boxplot represents the median of all input values. The box itself depicts 50 % and the whiskers are extended up to 1.5 times to the extreme points of the input values. If further values outside of these whiskers exist, the points are represented as outliers.

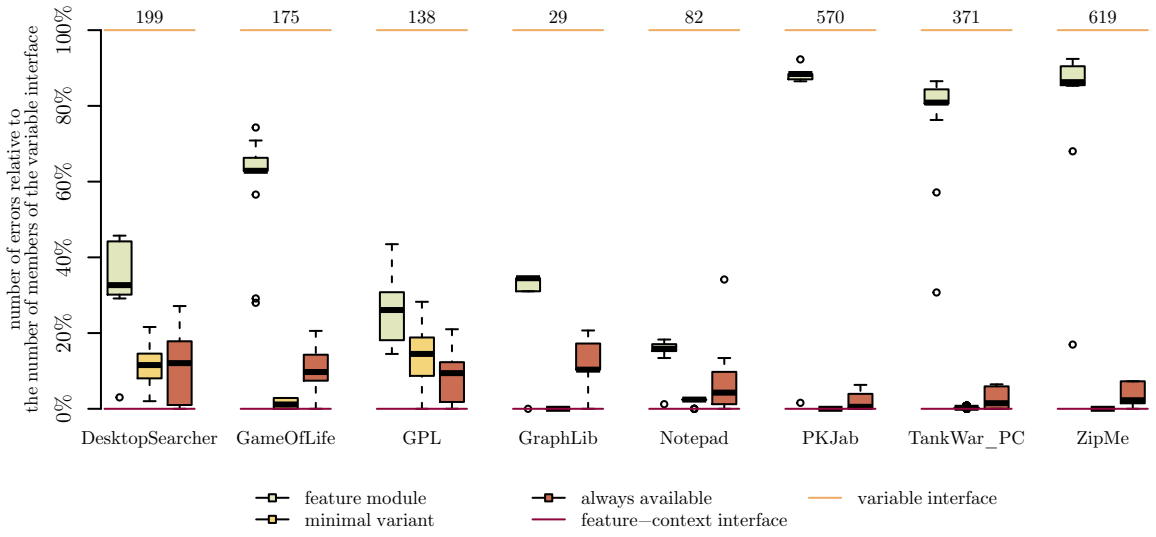


Figure 5.14: Potential errors for each state-of-the-art approach in comparison with the results of our feature-context interface. The results are scaled to the number of members in the variable interface.

using the approach *feature module* (lime-green box) are always equal or less than the number of API members of the approach *always available*. However, we can also see that the number of accessible API members using the approaches *minimal variant* (yellow box) and the feature-context interface (red box) are equal and noticeably larger than the number of API members given by the approach *always available*.

Besides the detailed outcomes of the product line *GraphLib*, we take a look at the general results of all product lines. Equal to the results of product line *GraphLib*, the product lines *ZipMe* and *PKJab* also present the same number of members for the approaches *minimal variant* and our feature-context interface. We observe similar results for the product line *TankWar_PC*. However, we see notable differences between the approaches feature-context interface and the *minimal variant* for the product lines *DesktopSearcher* and *GPL* whereas the results for *GameOfLife* and *Notepad* only present marginal advantages for the feature-context interface. Furthermore, the feature-context interface presents noticeably more API members in all cases than the approach *always available* and *feature module*.

To get some insights into the reasons of the similar results for the approaches feature-context interface and *minimal variant*, we take a look into the feature dependencies. Therefore, we also present the number of alternative features for each product line in Table 5.2. As a result, we found out that exactly the product lines with no alternative features (i.e., *GraphLib*, *PKJab*, *ZipMe*) present no differences in the approaches *minimal variant* and our feature-context interface. In addition, also the product line *GameOfLife* with only two alternative features presents almost identical results for both approaches.

Feature Context	*Minimal Variant*	*Feature Module*	*Always Available*
BasicGraph	0 (0.00)	0 (0.00)	0 (0.00)
Color	0 (0.00)	9 (0.31)	5 (0.17)
PrintHeader	0 (0.00)	10 (0.34)	3 (0.10)
Recursive	0 (0.00)	10 (0.34)	3 (0.10)
Weight	0 (0.00)	10 (0.34)	6 (0.21)

Table 5.4: Amount of errors of the product line *GraphLib* for each state-of-the-art approach based on a comparison of the accessible API members with our feature-context interfaces. The number in parentheses represents the values scaled to the number of members given in the variable interface.

Question (b): Potential Errors. In this section, we take a look at the results for question (b), the number of potential errors that can occur when using the state-of-the-art approaches. To this end, we use the same product lines as for question (a) with another viewpoint to the data behind them.

In Figure 5.14, we present the results for the investigation of potential errors. To ease comprehension, we also take a look at the details of product line *GraphLib*. In Table 5.4, we present the errors that a developer might make when applying the state-of-the-art approaches for each feature. This means, we compute the results for the feature-context interface, and the other state-of-the-art approaches for each concrete feature. Afterwards, we compare each API member of a state-of-the-art approach with each API member of the feature-context interface. If we cannot detect a corresponding member during this comparison, we count it as an error. For instance, the approach *feature module* for feature *Color* results in 9 missing API members compared to the feature-context interface and, thus, we count 9 errors. Similarly, we count 5 errors for the approach *always available*. In both scenarios, the missing API members reduce the reuse potential of this product line. However, if we compare the API members of the *minimal variant* with the API members of our feature-context interface, we detect no differences in all the features. Thus, in the case of the product line *GraphLib*, the approach *minimal variant* does not lead to any errors. Coming back to Figure 5.14, we use the feature-context interface as the base line (value 0) and scale all detected errors to the number of API members in the variable interface and use boxplots for the representation of these values.

If we take a look at the general outcome of this investigation, we can see similar results as for question (a). In detail, we can see that the approach *minimal variant* does not cause compile-time errors for the product lines *ZipMe*, *PKJab*, and *GraphLib*. However, for all other product lines, several compile-time errors are possible. Especially the product lines *GPL* and *DesktopSearcher* with a high amount of potential errors probably result in compile-time errors. We assume that these high values are the result of alternative dependencies in the feature model. Indeed the product line *GPL* has 15 (39 %) alternative features and the product line *DesktopSearcher* has 8 (36 %)

alternative features, whereas the percentage of alternative features for all other product lines is much lower.

Besides the outcome for the *minimal variant*, the outcome for the other approaches is also interesting. In Figure 5.14, we can see that also the approach *always available* produces a lot of errors and, thus, it also reduces the reuse potential of the product lines. Nevertheless, the approach *always available* seems to be more useful than the approach *feature module*, as this approach frequently creates a higher error rate.

Discussion

We now discuss the implications of our results for each approach and give suggestions in which scenario which approach is appropriate.

Feature Module. First of all, we take a look at the sound approach *feature module*. The quantitative evaluation has shown that the approach *feature module* presents only a small subset of all safely accessible API members. Thus, we have also seen that this behavior results in a huge amount of errors that reduces the reuse potential of the product line. In practice, we do not recommend to use this approach to find API members for the maintenance of a single product line. In the context of a multi product line, this approach is completely unable to identify any accessible members of a reused product line (cf. Section 5.1).

Always Available. Similar to the approach *feature module*, we also classified the approach *always available* as sound. However, compared to the approach *feature module*, this approach presents noticeably more accessible API members and, thus, we prefer the usage instead of the approach *feature module*. In addition, even if the approach *always available* also leads to a huge amount of errors that is often higher compared to the results of the approach *minimal variant*, we suggest to use the approach *always available* instead. This has two reasons. First, errors that reduce the reuse opportunities are less problematic than compile-time errors. Second, the approach is independent from a feature context and, thus, it is sufficient to compute it once per product line. Furthermore, in contrast to the approach *feature module*, we can also use this approach in the context of multi product lines to find safely accessible members of a reused product line.

Minimal Variant. Apart from the sound state-of-the-art approaches, we also investigated the complete approach *minimal variant*. Even if the approach presents partly the same results as the feature-context interface (i.e., without errors), we cannot recommend to use this approach to find safely accessible members. This has also multiple reasons. First, this approach can lead to compile-time errors and, thus, the developer needs additional effort to fix them. Second, we cannot predict in which cases the approach *minimal variant* presents good or bad results. Even if we know that alternative features are one reason of undesired results, it is not obvious for each feature model that a feature is an alternative feature. For instance, if the feature is an alternative feature because of cross-tree constraints, it is possible that a developer is not able to detect this

feature dependency without tool support. Third, we know from other experiments that alternative features in feature models are not the exception [Thüm et al. 2009]. Thus, the probability of compile-time errors using the approach *minimal variant* is increased. Therefore, we cannot recommend to use this approach to find safely accessible API members, neither in single product lines nor in multi product lines.

Feature-Context Interface. With our evaluation we have shown that all state-of-the-art approaches can lead to compile-time errors or that they reduce the reuse potential of a product line. As the feature-context interface always presents the complete set of safely accessible API members that can be used in an implementation, maintenance, or development step, we recommend to use feature-context interfaces to solve these tasks. For the case of a single product line, we have to determine the variable interface and the feature-context interface on the fly, as the feature context and the implementation artifacts will change continuously as the developer shifts his/her focus from one feature to another and adds, removes, or changes functionality. However, especially in the case of multi product lines (cf. Section 5.4.2), in which the program artifacts of a reused product line are stable, we recommend to use feature-context interfaces. In detail, in a scenario in which we plan to reuse another product line, we can precompute the variable interface so that it is only necessary to compute the feature-context interface according to a feature context on the fly.

General Results. According to our investigation of Hypothesis 2 we found out that the variable interface in combination with the feature-context interface can help the developer to detect reusable implementation artifacts. In detail, the variable interface can help to get an overview of all API members of all features so that we can find the appropriate API member for a maintenance task more efficiently as with state-of-the-art approaches. However, only in combination with feature-context interfaces, we get the real potential of the variable interface to find safely accessible API members for a maintenance or development task. Even if this evaluation focused on a single product line, the application of the variable interface with feature-context interfaces in a multi product line is similar. Furthermore, the variable interface is more important in a scenario of multi product lines. For this application scenario, we also presented filtering techniques so that the variable interface only contains API members that we can use in combination with our features of interest (i.e., features of the feature-model interface). All other features are hidden from the developer's perspective.

5.5.3 Threats to Validity

We now discuss internal and external threats to validity.

External Validity. For the discussion of the external validity, we consider the generalizability of our evaluation results. Of course, we only used product lines written in Java with FOP, which reduces the result's generalizability according to other languages and paradigms. However, to ensure that our results are generalizable for our use case, we used a set of different systems from multiple domains and with different sizes. Furthermore, we used product lines that were also used in other papers for the purpose of

evaluation (cf. [Apel et al. 2013b; Kolesnikov et al. 2013; Thüm et al. 2011a]). Thus, we assume that we investigated a set of product lines that are accepted by the product-line community.

Internal Validity. To discuss the internal validity, we take a look into the activities that we used to minimize the probability of systematic errors. One critical aspect could be an already existing compile-time error in the investigated product lines. These errors can influence the collection and comparison of API members, and, thus we decided to only use product lines without compile-time errors for our evaluation. Another critical aspect addresses the correct collection and comparison of API members. For these tasks, a string comparison or an ad-hoc parser implementation can be error-prone and, thus, we used FUJI for this purpose. In detail, FUJI is a Java compiler and a type checker for feature-oriented programming [Apel et al. 2012; Kolesnikov et al. 2013]. To this end, we rely on FUJI for the collection and comparison of fields and methods. However, we also want to clarify that we do not use the common part of the Java-field and Java-method signature for our comparison. For instance, in Java, a method signature only consists of the method name and the parameter list. In contrast, we also have to compare the return type of the method since different features can use different return types.

5.6 Summary

In this chapter, we considered the syntactical interface as second interface of our concept of multi-level interfaces. In contrast to the variability-model interface for the modeling level of multi product lines, the syntactical interface can be used to support the implementation of multi product lines. In detail, the syntactical interface presents an overview of all API members that a developer can reuse from a product line in another product line. To investigate the concept, we implemented a syntactical interface for product lines based on the programing language Java with the paradigm of FOP. We called the result a variable interface that presents all API members of a product line with additional variability information. Depending on the application scenario, we can filter the variable interface accordingly. For this purpose, we introduced two filter approaches that allow us to create a tailored variable interface according to the features of interest given by the feature-model interface. As the filtering result, the developer gets an overview of all accessible API members of a product line he/she can reuse.

Indeed, the filtered variable interface hides all API members that a developer cannot reuse and, thus, it can ease the development of a multi product line. However, as the variable interface also contains variability information, it is still a manual effort to identify safely accessible API members for a specific implementation task. Therefore, we also introduced the feature-context interface that is a non-variable view on the variable interface tailored to the current needs of the developer. Besides the usage of this approach for multi product lines, it can also be used for single product lines. Even if some state-of-the-art approaches exist to identify accessible API members for an implementation task, we discussed that only feature-context interfaces are able to present a complete view on safely accessible API members.

To investigate whether the variable interface and the subsequent feature-context interfaces can help to develop (multi) product lines in practice (cf. Hypothesis 2), we also performed an evaluation. In detail, we compared the API members given by the feature-context interfaces with the results of the state-of-the-art approaches. As the result, we found that only feature-context interfaces are able to present a sound and complete overview of safely accessible API members. In contrast, the usage of presented API members given by state-of-the-art approaches can lead to compile-time errors or to a reduced reuse potential of a product line.

6. The Behavioral Interface - The Third Level of Multi-Level Interfaces

> The chapter about behavioral product-line interfaces shares material with our overview paper *Towards Modular Analysis of Multi Product Lines* [Schröter et al. 2013a]. In this paper, we present the behavioral interface as part of our concept of multi-level interfaces. Based on these initial ideas, we extend and improve the concept in our paper on *Variability Hiding in Contracts for Dependent Software Product Lines* [Thüm et al. 2016]. Therefore, the chapter also shares material with this investigation and evaluation of behavioral product-line interfaces.

In this chapter, we consider the behavioral (product-line) interface as proof of concept regarding the multi-level interfaces and investigate whether it is possible to extend the multi-level interfaces to further levels, such as the specification and verification of multi product lines. In detail, we now consider the behavioral product-line interface that can be used to ensure a correct communication between dependent product lines. Therefore, the behavioral product-line interface is an agreement on the behavior of interacting elements (e.g., API members) between two product lines, whereas the concrete implementation and not required features of the reused product line are hidden (cf. a feature-model interface that only presents features of our interest). This information can be used to verify the behavior of the product line that aims to reuse it. Since each interface on a lower level of our concept of multi-level interfaces depends on the interfaces of the upper levels, our behavioral product-line interface depends on the concrete implementations of the variability model interface and the syntactical product-line interface. As a result, the behavioral product-line interface only considers elements that are represented in these upper-level interfaces. Thus, the behavioral product-line

interface also aims to remove the direct dependencies between the dependent product lines and to ease the verification and evolution of the multi product line.

In the previous chapters, we exemplified the variability model interface by the feature-model interface and the syntactical product-line interface using the variable interface with the feature-context interface. Therefore, our behavioral product-line interface depends on these concrete implementations of our upper-level interfaces. In detail, the behavioral product-line interface needs to define the behavior of the API members given by the variable interface that again only presents members according to the features of interest represented in the feature-model interface. For the specification of the behavior of API members, we use feature-oriented contracts (cf. Section 2.1.3 for a brief overview, or the thesis by Thüm for more details [Thüm 2015]), because the concrete implementation of our previous interface (i.e., variable interface) is based on FOP. Using this concrete scenario, we investigate our Hypothesis 3 and want to know whether the behavioral product-line interface enables a time-efficient modular analysis to detect violations in multi product lines using verification techniques.

To illustrate the dependencies of the behavioral product-line interface to the other interfaces, we take a look at our running example. In detail, we are interested in the features *BankAccount* (B), *DailyLimit* (D), *Interest* (I), and *SimpleLock* (S) of product line *BankAccount*, as they are needed to implement the features of the product line *BankApplication*. Therefore, the feature-model interface consists of these features and the variable interface with the feature-context interfaces only presents members that are available if we use these features (cf. Chapter 5). As a result, the behavioral product-line interfaces need to define feature-oriented contracts for members given in the filtered variable interface so that it is possible to verify the correct behavior of methods in the product line *BankApplication* (i.e., to verify method `transfer`) without further information about the hidden product line *BankAccount*.

The chapter is structured as follows. First, we give an overview of a straight-forward verification of multi product lines using design by contract. Second, we discuss different strategies that we can use to achieve a behavioral product-line interface. Third, we investigate the verification effort for the multi product line *BankApplication* using our introduced strategies for a behavioral product-line interface between the product lines *BankApplication* and *BankAccount*. Finally, we summarize our findings.

6.1 Specification and Verification Based on Feature-Oriented Contracts

In this section, we illustrate how to specify and verify multi product lines in a straight-forward manner based on feature-oriented contracts [Thüm 2015] and variability encoding [von Rhein et al. 2016] and consider the problems of this application strategy. In general, the specification and verification process of a multi product line can be performed in a similar manner as for a single product line. Therefore, we recapitulate our illustration of Section 2.1.3 and give a brief summary on how to specify and verify a

product line. Afterwards, we consider the application of this procedure in a scenario of a multi product line using our running example of product line *BankApplication* with product line *BankAccount*.

As described in more detail in Section 2.1.3, it is possible to use an extended version of design by contract and variability encoding to ease the specification and verification of a product line. In detail, we use feature-oriented contracts as an extension of design by contract for FOP that allows a developer to refine existing specifications (e.g., using the keyword `original`) [Thüm 2015; Thüm et al. 2012]. Afterwards, we can use variability encoding in which the compile-time variability is translated into run-time variability [von Rhein et al. 2016]. As a result, the product line is represented in one product called metaproduct that includes the functionality of all product-line's products. At the same time, we can also use variability encoding to translate the feature-oriented contracts into this metaproduct so that a method contract considers all desired feature selections [Thüm et al. 2014; Thüm et al. 2012]. Since the outcome is a regular JML expression, we can use state-of-the-art tool support for product verifications. To ensure that only valid feature combinations are verified (i.e., all other can lead to errors), the dependencies given in the feature model are additionally represented as an invariant in each class. The invariant demands the feature dependencies to hold, before and after each method execution. Thus, if we use the resulting metaproduct for the purpose of verification, we can verify all valid products of the product line in one step.

Similar to the verification process of a single product line, we can also apply the same concept to multi product lines. Therefore, we need to consider the multi product line as one single product line, in which we have access to all variability information, implementation artifacts, and contracts. As a result, we can generate a metaproduct that encodes the complete variability of the composed product lines. Afterwards, we can use state-of-the-art tool support for the program verification. However, this procedure is inefficient, as it is necessary to reapply the process if one of the involved product lines changes. Therefore, it doesn't matter if a product-line adaption occurs in the source code, in the contracts, or even in a feature model. In all of these cases, it is necessary to execute the verification process again.

Running Example

Using our running example, we illustrate the verification process of a multi product line. In Figure 6.1, we present an excerpt of our product line *BankApplication*. In detail, we depict the method `transfer` of the class `Transaction` that is used to transfer money from one account of product line *BankAccount* to another. To ensure the correctness, the method also defines contracts. In these contracts, the `requires` clauses cause the source and destination account to be not `null` and not equal (cf. Lines 1, 2). Furthermore, the `ensures` clauses make sure that the amount of money is removed from the source and added to the destination if the transfer was successful. Otherwise, it ensures that the balance of both accounts is equal to the state before the method execution was started (cf. Lines 3–6).

```
1 /*@ requires dest != null && src != null;
2   requires src != dest;
3   ensures \result ==> (\old(dest.balance) + amount == dest.balance);
4   ensures \result ==> (\old(src.balance) - amount == src.balance);
5   ensures !\result ==> (\old(dst.balance) == dst.balance);
6   ensures !\result ==> (\old(src.balance) == src.balance); @*/
7 boolean transfer(Account src, Account dest, int amount) {
8   if (!lock(src, dest)) return false;
9   try {
10    if (amount <= 0) return false;
11    if (!src.update(-amount)) return false;
12    if (!dest.update(amount)) {
13      src.undoUpdate(-amount);
14      return false;
15    }
16    return true;
17  } finally { src.unLock(); dest.unLock(); }
18 }
```

Figure 6.1: Method `transfer` of class `Transaction` from the product line *BankApplication* with contracts.

To verify the method `transfer` of product line *BankApplication*, we have to create the metaproduct. However, the method's behavior depends on the selection of features of the product line *BankAccount*. For instance, the method `transfer` calls the method `update` (cf. Lines 11, 12) that is implemented in the class `Account` of the product line *BankAccount*. Depending on the feature selection, the method `update` can have a different behavior and, thus, different contracts that also have an impact on the method `transfer`. To verify the correctness of all possible products for the dependent product line *BankApplication*, we need to create the metaproduct of the composed product lines. For this purpose, we also need to describe all feature dependencies in a JML invariant so that we only verify valid products. Therefore, we define all feature variables (i.e., $\mathcal{F}_{BankApplicationt/BankAccount}$) in a separate class `FM` (cf. Figure 6.2, Lines 1–3) and use these variables in all classes of the multi product line to describe the run-time variability and the specifications (e.g., the invariant). For instance, the class invariant of class `Transaction` represents the complete feature dependencies of the composed feature model (i.e., logical representation of the composed feature model with all features - $\mathcal{M}_{BankApplication/BankAccount}$). In Figure 6.2, we present parts of this invariant (cf. Lines 5–7). In detail, Line 5 causes the feature *BankApplication* to be always available, whereas the subsequent line ensures the child-parent implication between the features *BankAccount* and *BankApplication* (cf. Line 6). However, the most significant dependency for the dependent product line is given in Line 7, as it describes the inter-model dependency between the features *Transaction* and *SimpleLock*. In sum, using this invariant, the metaproduct encodes the complete dependencies of the composed product lines. Afterwards, we can use the state-of-the-art tool support to verify the method's correctness.

In summary, the described verification process of method `transfer` of the multi product line *BankApplication* needs access to all dependencies of the complete feature-model composition with all feature modules and contracts of product line *BankApplication*

```
1public class FM {
2  public static boolean bankApplication, transaction, bankAccount, simpleLock [, ...];
3}
```

```
4public class Transaction{
5/*@ invariant FM.bankApplication &&
6                    (FM.bankApplication || !FM.bankAccount) [&& ...]
7                    (FM.simpleLock || !FM.transaction) [&& ...];@*/
8
9  /*@ [contracts of method transfer cf. Figure 6.1] @*/
10 boolean transfer(Account src, Account dest, int amount) {
11   //implementation of method transfer cf. Figure 6.1
12 }
13}
```

Figure 6.2: Method `transfer` of class `Transaction` from the product line *BankApplication* and an invariant that represents the feature dependency of the complete feature-model composition.

and product line *BankAccount*. As we are only interested in reusing parts of the product line *BankAccount*, this is an undesired effort. In detail, the method `transfer` needs the knowledge of contracts from all methods that are called (e.g., method `update`) for the verification. However, these contracts are based on features that are not of our interest. We assume that this leads to more effort in the verification and can also result in drawbacks regarding the verification performance. Therefore, we propose to use our concept of multi-level interfaces with the behavioral product-line interface to focus on the features of interest and to reduce the verification effort.

6.2 Strategies for the Representation of the Behavioral Interface

In this section, we give an overview of the different strategies to achieve a behavioral product-line interface. In detail, we assume that the initial product line *BankAccount* is based on FOP with feature-oriented contracts and that the feature-model interface already exists. Based on this assumption, each strategy aims to create the metaproduct in a way that only features of interest from our feature-model interface are used in the implementation and contracts. We only investigate strategies that, in theory, allow us to automatically generate the behavioral product-line interface in future because manually created interfaces result in too much effort and mitigate a possibly reduced verification time. As each concept comes with limitations, we present benefits and drawbacks of each strategy and investigate their applicability in our running example. In the following, we present the strategies *false configuration*, *true configuration*, and *hidden configuration*.

6.2.1 False Configuration

The first strategy is called *false configuration*, as we bind each feature that is not in the feature-model interface to `false`. Therefore, it is similar to a feature-model configura-

```
1class Account {
2  //@ requires daysLeft >= 0;
3  //@ ensures calculateInterest() >= 0 ==> \result >= interest;
4  /*@ pure @*/ int estimatedInterest(int daysLeft) {
5    return interest + daysLeft * calculateInterest();
6  }
7}
```

(a) Method `estimatedInterest` in feature module *InterestEstimation*.

```
8class Account {
9  //further source code of the metaproduct
10
11  //@ requires FM.interestestimation && daysLeft >= 0;
12  //@ ensures calculateInterest() >= 0 ==> \result >= interest;
13  /*@ pure @*/  int estimatedInterest(int daysLeft) {
14    return interest + daysLeft * calculateInterest();
15  }
16}
```

(b) Method `estimatedInterest` in the metaproduct of product line *BankAccount*.

Figure 6.3: Method `estimatedInterest` of class `Account` of product line *BankAccount* with contracts from the feature module *InterestEstimation* and of the metaproduct of product line *BankAccount*.

tion (cf. Section 2.1.1), in which we deselect all features that are not in our focus (i.e., features that are not part of our features of interest). Based on this idea, we create a metaproduct of the product line and, thus, transform the compile-time into run-time variability with these features only. In detail, compared to an ordinary metaproduct (cf. Section 2.1.3), the resulting metaproduct of the strategy *false configuration* contains adaptions in the source code and the contracts. The source-code adaptions result in unreachable and, thus, ignored branches, as the replacement of the corresponding features with `false` automatically removes the alternative behavior. In addition, the strategy also adapts invariants, pre- and postconditions in which features exist that are not of our interest. A subsequent simplification of all contracts can lead to a false evaluation of a method's precondition and, thus, we can remove the complete method as it will not be allowed to access it.

To illustrate the strategy *false configuration*, we use our running example of product line *BankAccount*. In Figure 6.3(a), we depict the method `estimatedInterest` of the feature module *InterestEstimation* with feature-oriented contracts. As described in the precondition, a valid execution of the method requires that the parameter `daysLeft` is positive. Based on this condition, the method ensures that, if the calculated interest is positive, the result will be greater than the value stored in the field `interest`. If we create the metaproduct of this product line, the contract will be automatically transformed so that the precondition also requires that the feature *interestEstimation* is `true` (cf. Line 11 of Figure 6.3(b)). However, using the strategy *false configuration*, the feature *interestEstimation* will be set to false and, thus, the complete precondition can never become true. As a result, it is not allowed to call this method.

Even if it is possible to apply this strategy to our running example, the strategy *false configuration* is not applicable to all scenarios. In fact, if the features that we want to remove from the contract lead to a contradiction according to the feature-model dependencies, the application of the strategy is not possible. For our running example, the set of features that are not of our interest do not lead to a contradiction of the feature-model dependencies given in the invariant. However, if we assume to focus on feature *InterestEstimation* but not on feature *Interest*, it is necessary to replace feature *Interest* with `false`. According to the feature-model dependencies, we also need to replace feature *InterestEstimation* by `false` to be conform to the feature-model dependencies. Hence, this replacement contradicts our assumption in which we are interested in feature *InterestEstimation*. Thus, it is not possible to remove an arbitrary set of features using the strategy *false configuration*.

6.2.2 True Configuration

Our second strategy named *true configuration* is similar to *false configuration* but binds the corresponding features to `true`. In detail, we also use this configuration to create a metaproduct. The compile-time variability is transformed to run-time variability, in which the branches of features that are not of our interest are automatically activated in this resulting metaproduct. Furthermore, all occurrences of these features in contracts or invariants are replaced by `true` and simplified.

For the purpose of illustration, we use our product line *BankAccount*. Here, we are interested in the root feature *BankAccount*, and in the optional features *DailyLimit*, *Interest*, and *SimpleLock*. Using the strategy *true configuration*, we bind all other features of the metaproduct to `true`. Even if this is possible for our running example, it leads to some special characteristics. First, if we would bind the feature *InterestEstimation* to `true`, the feature is included in all products. As a result, the feature *Interest* as parent feature of *InterestEstimation* has to be also bound to `true`, to be compatible to the feature-model dependencies. Therefore, the feature *Interest* acts as mandatory feature and, thus, it reduces the variability. Second, a complex problem is the alternative group below feature *Lock*. In this scenario, we have to bind the feature *TimeUnitLock* to `true` and, thus, it is not allowed to select the feature of interest *SimpleLock*. In a final product-line implementation, this is a contradiction and would prohibit an application of the strategy *true configuration* to our running example. Fortunately, it exists only a plan to implement the features *Lock* and *TimeUnitLock* in future (cf. Figure 2.1 (a) of Section 2.1.1) and, thus, both features are abstract features without code artifacts. This characteristic directly influences the representation of the invariant, in which the feature dependencies are described. In detail, as the invariant needs to describe dependencies of concrete features (i.e., with code artifacts), there is no reference to the features *Lock* and *TimeUnitLock*. Therefore, at the running example of product line *BankAccount*, it is still possible to apply the strategy *true configuration* and bind all concrete features to `true`.

In sum, the application of the strategy *true configuration* is restricted. Even in our small running example, the application of the strategy is only possible with some limi-

tations as the feature *Interest* will be mandatory and reduces the intended variability. Nevertheless, it is a simple strategy to remove features from contracts and invariants to create a behavioral product-line interface.

6.2.3 Hidden Configuration

The third strategy to create a behavioral product-line interface is called *hidden configuration*. In contrast to the previous strategies, the strategy *hidden configuration* does not bind features that are not of our interest, the strategy considers both cases (i.e., `true` and `false`). Therefore, the strategy is called *hidden configuration*, as it covers existing variability of the product line but it does not remove the conditions of contracts. The strategy *hidden configuration* is the most complex strategy of our investigation. In detail, we have to create the metaproduct and need to transform all pre- and postconditions so that both cases, `true` and `false`, are considered. In contrast to the other strategies, in which a feature is bound to `true` or `false` so that also the source code is affected (cf. removed branches of the other strategies), the strategy *hidden configuration* does not change the method body. However, it is necessary to treat the pre- and postcondition in a different manner, as we will show in the following.

As the adaption of the postcondition for the strategy *hidden configuration* is similar to the creation of the feature-model interface (cf. Chapter 4), we start to consider the postcondition first. In detail, the ensures clauses of the metaproduct represent the different results of the method with and without a specific feature that we want to remove from the condition. Therefore, we want to keep this information but want to remove the dependency to the specific feature. To achieve this outcome, we use the same principle as for the generation of the feature-model interface. First, we combine all ensures clauses so that we have a condition that represents the complete output behavior. Second, we duplicate the resulting formula of the ensures clause and replace all occurrences of the specific feature in the first formula with `true` and in the second formula with `false`. Third, we connect both formulas by a disjunction. Thus, the resulting formula ensures a specific output behavior of the method if the feature is `true` and also if the feature is `false`. After this procedure, we can simplify the formula and we can repeat this procedure to remove a second feature that is also not of our interest.

As the precondition of a method exists to specify under which input condition the output represented in the ensures clause can be ensured, we have to fulfill both possible selections (i.e., `true` and `false`). As a consequence, we also duplicate the existing formulas and replace the feature with `true` in the first and with `false` in the second formula. In contrast to the postcondition, we combine both formulas by a conjunction to create a new precondition. Without this conjunction, it is possible to choose a product in which the precondition is violated. To remove another feature from this precondition, we repeat the procedure.

To illustrate the application of the strategy *hidden configuration*, we use our running example of the product line *BankAccount*. In detail, we plan to remove feature *Overdraft*, as it is not of our interest. In Figure 6.4(a),(b), we present method

`getOverdraftLimit` of class `Account` and the postcondition's transformation. In detail, Figure 6.4(a) presents the result of the first step in which we combined the two existing postconditions (cf. Figure 2.10) to one postcondition. Afterwards, we duplicate the formula and replace all occurrences of overdraft with `true` in the first and `false` in the second part of the formula. The resulting formulas are combined by a disjunction (cf. comments of Lines 14–15). The final result of the transformation is represented as a simplification of this formula, in which the feature *Overdraft* is completely removed. As a result, the postcondition does not depend on the feature *Overdraft* and, thus, it is ensured that the method returns either `-5000` or `0`.

In summary, the strategy *hidden configuration* fits best to our needs of a behavioral product-line interface, as it hides the variability and does not bind it. Thus, several problems of the other strategies, such as the necessary selection of features that are not of our interest (see strategy *true configuration*), are not possible. However, the conjunction in the precondition can lead to formulas that are not satisfiable. In our running example this situation does not occur, as it is only a problem if the specific method is called. In addition, as we will also discuss in the evaluation, the postcondition can be too weak to verify callers of the method. For this case, we can choose another strategy or we can add further features of interest.

6.3 Evaluation: The Interface Strategies as Behavioral Interface in Practice

In this section, we present the evaluation of our strategies for a behavioral product-line interface. In detail, we want to investigate Hypothesis 3 and, thus, we plan to compare the verification effort of a multi product line with each of our proposed interface strategies and without an interface. According to Hypothesis 3, we are especially interested in performance benefits and want to determine and compare the time for each verification.

Besides the investigation of the time-efficiency of the verification process, a second goal of our evaluation is to find out, whether a strategy is appropriate for an implementation of a generation strategy. In detail, we assume that it is possible to generate the specification of each strategy in a way that it is not necessary to verify the interface in future. However, in this evaluation, we use a manual transformation into our interface strategies. If we find out that an interface presents promising results regarding the verification effort, we can spend more effort in an automatic generation strategy and in proofs for their correctness.

For the purpose of illustration, we take a look at our running example. Using a manual transformation, we can create an interface for each strategy of product line *BankAccount* that is reused from product line *BankApplication*. Based on these implementations, we are interested into the verification effort for product line *BankApplication* in four different scenarios. First, using the complete information of contracts in the metaproduct of product line *BankAccount*. Second, using the simplified metaproduct and contracts of each strategy from the interfaces of product line *BankAccount*.

```
1class Account {
2  //further source code of the metaproduct
3
4  //@ ensures !FM.overdraft ==> \result == 0 && FM.overdraft ==> \result == -5000;
5  int /*@ pure @*/ getOverdraftLimit(){
6    if (!FM.overdraft) return 0;
7    return -5000;
8  }
9}
```

(a) Method `getOverdraftLimit` with combined postconditions in the metaproduct of product line *BankAccount*.

```
10class Account {
11  //further source code of the metaproduct
12
13  //intermediate result of the strategy hidden configuration:
14  //    ((!true || \result == -5000) && (true || \result == 0)) ||
15  //    ((!false || \result == -5000) && (false || \result == 0));
16
17  //final result of the strategy hidden configuration with a simplified postcondition:
18  //@ ensures (\result == -5000) || (\result == 0);
19  int /*@ pure @*/ getOverdraftLimit(){
20    if (!FM.overdraft) return 0;
21    return -5000;
22  }
23}
```

(b) Method `getOverdraftLimit` with a simplified postcondition after the application of strategy *hidden configuration*.

Figure 6.4: Method `getOverdraftLimit` of class `Account` represented in the metaproduct before and after the application of the strategy *hidden configuration*.

In the following, we present our experiment design and results. Afterwards, we present open challenges for future work.

6.3.1 Experiment Design

In this section, we introduce our subject system and present details on the experiment design for the investigation of our strategies for a behavioral product-line interface.

Subject System

According to our experiment idea, we want to investigate the verification effort of a multi product line in different scenarios. As the specification of an object-oriented system and especially of product lines and multi product lines is a complex task, we use a special variant of the product line *BankApplication* with an already existing specification for our evaluation. Furthermore, the product line was used in multiple studies and, thus, it is already known in the community [Thüm 2015; Thüm et al. 2014]. However, in contrast to our running example, the original product line *BankApplication* is not a multi product line but a single product line. In a first step of the experiment, we decomposed this single product line into a representation that is similar to our running example so that

the product line *BankApplication* reuses the product line *BankAccount*. Basically, the resulting product lines of the decomposition (i.e., *BankAccount* and *BankApplication*) are in parts different to our running example. First, the product line *BankAccount* holds an additional concrete and optional feature named *Logging* that allows us to log all changes of the bank account and, thus, the feature refines already existing methods of the class `Account`. Second, the dependency between both product lines is different. In the investigated scenario, the bank account is part of all applications. For this reason, the root feature *BankAccount* is mandatory and not optional as in all other descriptions. Finally, the product line *BankApplication* also holds implementation artifacts for the features *BankAccount*, *DailyLimit*, and *Interest*.

Experiment Execution

The experiment is designed as follows. First, we decompose the product line *BankApplication* into the two dependent product lines (i.e., product line *BankApplication* and product line *BankAccount*, see also the description above). As a result, the product line *BankAccount* contains the implementation artifacts and refinements of class `Account`, whereas product line *BankApplication* contains the classes `Transaction` and `Application`. The reason for this kind of decomposition is based on the used theorem prover KeY and the knowledge that existing proofs retain their validity [Beckert et al. 2007]. Second, we manually create the interfaces for product line *BankAccount* (i.e., using the strategies *false*, *true*, and *hidden configuration*). Third, we use KeY for our proofs. We start to prove the correctness of product line *BankAccount* using the original contract definition. Furthermore, we also prove the correctness of all three interfaces against the implementation of product line *BankAccount*. This allows us to ensure the correctness of these interfaces. In future, these proofs based on KeY are not necessarily needed if we can prove that the interface generation does not break any existing proofs. Then, we verify product line *BankApplication* using the contracts of each interface strategy. Finally, we verify the product line *BankApplication* using the original contracts of product line *BankAccount*. For the execution of all proofs, we configure KeY so that *method treatment* is based on contracts. In contrast, if method treatment allows inlining instead, the proofs also depend on the implementation of called methods. However, this is not compatible to our interface concept and we rely on the contracts instead. For all verifications, we used a notebook with Intel Core i7 (2.4GHz), 8 GB RAM, and Windows 7, to determine the number of nodes, the time, and the needed proofs. Furthermore, as the strategies *true* and *false configuration* bind the variability, we also consider the number of products for which the specific proof ensures the correctness.

6.3.2 Experiment Results and Discussion

Based on the decomposition of product line *BankApplication* with the classes `BankApplication` and `Transaction`, and the product line *BankAccount* with the class `Account`, we start our experiment and use this section to present our results. First,

		Interface Strategies		
	Contracts of *BankAccount*	False Configuration	True Configuration	Hidden Configuration
Nodes	296,961	37,134	242,699	38,460
Time (min)	13.6	1.95	15.9	1.44
Proofs	18	14	18	14
Products	96	8	4	96

Table 6.1: Verification effort of product line *BankAccount* and each interface strategy.

we present the results of the verification of product line *BankAccount* and each interface strategy. Second, we present the verification effort of the multi product line *BankApplication* using the original contracts of product line *BankAccount* and the contracts of each interface strategy. Afterwards, we discuss our results.

Verification Results of Product Line BankAccount and all Interfaces

In Table 6.1, we present the result of our experiment regarding the product line *BankAccount* and its behavioral product-line interfaces. The verification of the product line *BankAccount* with original contracts needs 296,962 nodes, 13.6 minutes and 18 proofs. If we compare these results with the interface strategies, we can see that all strategies are able to reduce the number of nodes to close the proofs. Furthermore, the strategies *false* and *hidden configuration* significantly reduce the number of proofs and the time that is necessary for each proof. The number of proofs is reduced as we removed all methods with a *false* precondition. Since it is not allowed to call these methods, a proof of the method correctness would be useless. In contrast to the other interface strategies, the strategy *true configuration* does not reduce the number of proofs and the time of the proofs is also slightly increased. We also present the number of products for which each strategy ensures the correct behavior. In detail, the original contracts ensure the correct behavior of 96 products. Using the strategy *hidden configuration*, we also can ensure the correct behavior of 96 products, whereas the adaption of the metaproduct for the strategies *false* and *true configuration* only ensures the correctness of 8 and 4 products.

Verification of Product Line BankApplication

Using the contracts of product line *BankAccount*, we now verify the product line *BankApplication*. In Table 6.2, we present the verification effort of product line *BankApplication* using the original contracts of product line *BankAccount* and the different interface strategies. First, we investigate the verification with the original contracts, in which KeY needs 95,292 nodes, and 5.37 minutes for all 11 proofs. Second, we also verified all methods of product line *BankApplication* using our interface strategies. Thus, the number of proofs is identical in all the verifications. Furthermore, all strategies

		Interface Strategies		
	Contracts of *BankApplication*	False Configuration	True Configuration	Hidden Configuration
Nodes	95,292	27,016	53,983	21,393
Time (min)	5.37	0.770	2.20	0.585
Proofs	11	11	11	11
Products	144	12	6	144

Table 6.2: Verification effort of product line *BankApplication* using the original contracts of product line *BankAccount* and the contracts of each interface strategy.

significantly reduce the time and the necessary nodes for the verification. The strategy *hidden configuration* presents the best results. In detail, the time is reduced by 89% and the number of nodes by 78%. At the same time the strategy *hidden configuration* allows us to ensure the correct behavior of all 144 products that we can also prove using the original contracts of product line *BankAccount*. However, the strategies *false* and the strategy *true configuration* also present significant improvements regarding the time and necessary nodes for all 11 proofs. In contrast to the strategy *hidden configuration*, these strategies cannot ensure the correct behavior of all 144 products. In detail, the strategy *false configuration* can be used to prove the correct behavior of 12 products whereas the strategy *true configuration* can only be used to investigate 6 products.

Discussion

To investigate Hypothesis 3, we verified the multi product line *BankApplication* that we decomposed from an already existing case study of a single product line for bank accounts into our dependent product lines. In detail, our investigation considered three different strategies for a behavioral product-line interface and we found that each of these strategies allows us to save time when verifying the product line *BankApplication*.

For the experiment, we assumed that the interfaces are correct by construction (i.e., constructed from a verified product line) and it is not necessary to verify them. As it is an open question if it is possible to prove this assumption, we reverified each interface strategy to ensure their correctness. However, we found, that even if our assumption is wrong, it is also possible to reverify the behavioral product-line interface in an appropriate time. Thus, even with these additional proofs, we can also save time for the verification of the multi product line *BankApplication*.

For the verification of multi product line *BankApplication*, we found that the strategy *hidden configuration* presents the best results compared to a verification using the original contracts of product line *BankAccount*. But also the strategies *false* and *true configuration* present significant improvements in time and needed nodes to close the proofs. However, the strategy *true configuration* presents the worst results. The reason for this characteristic is due to the variability binding. In detail, the strategy

true configuration includes the contracts of all features that we removed from the feature-model interface. Therefore, the complexity of the contracts will be increased and, thus, the verifications based on these contracts can lead to a time-consuming task. In contrast, the strategy *hidden configuration* seems to be the optimal solution for the representation of a behavioral product-line interface. However, in a special case the preconditions of this strategy can be too strong and the postcondition too weak. Although this is not the case in our subject system, we are confident that this situation can happen in other product lines. As a solution, it is possible to switch to another strategy. In contrast, we suggest to add additional features to the interface as the other strategies are based on the drawback of variability binding. Furthermore, another drawback is that the other strategies cannot verify all products of the multi product line.

As mentioned above, the strategy *hidden configuration* fits best to our concept of multi-level interfaces. In contrast to the strategies *false* and *true configuration*, it does not bind variability; it hides variability like the feature-model interface and the feature-context interface. At the same time the strategy leads to the best results in our experiment. To achieve the full potential of this strategy, we need to provide an algorithm to automatically derive the interface. If it is also possible to prove that this generated interface does not break already existing proofs, we can save the time that is necessary to reverify the interface. However, this is part of future work.

6.3.3 Open Challenges

We identified several challenges when analyzing the strategies of variability hiding for contracts. In the following, we discuss the challenges regarding *class invariants*, *framing conditions*, and *alternative values*.[6]

Class Invariants

In our experiment, we focused on method contracts for the specification and verification of the involved product lines. If several contracts of methods are identical, it is also possible to use *class invariants* to describe the desired conditions. As a result, the conditions of the class invariants need to hold before and after a method is executed. This can save time for the specification, as the condition needs to be described only once. Furthermore, the concept of invariants supports subtyping so that the invariant of a subtype needs to fulfill the condition of the supertype.

Considering class invariants, the question arises whether our strategies are also applicable to this kind of specifications. For the strategies *false* and *true configuration*, it is straight forward to apply the specific strategy. To illustrate the effect, we use a simple example of a feature f that introduces a class invariant *inv*. Furthermore, we assume that the feature f is not part of the feature-model interface and, thus, we want to remove the feature from the invariant. Using variability encoding, this definition results

[6] In addition to these challenges regarding our strategies, we also experimented with Java interfaces and abstract classes to achieve variability hiding. For more details, we refer the reader to the description of our corresponding paper [Thüm et al. 2016].

in an invariant $f \rightarrow inv$, so that the invariant has only an impact if the feature f is chosen. Applying strategy *false configuration* leads to $false \rightarrow inv$ which is a tautology. Therefore, the invariant *inv* has no effect and can be removed. In contrast, if we apply strategy *true configuration*, we replace f by `true` and, thus, the invariant is available in all configurations.

In contrast to the strategies *true* and *false configuration*, the application of the strategy *hidden configuration* to invariants is more complicated. At the moment, we have no elegant solution for the strategy *hidden configuration*. As a simple solution, we removed each invariant in our subject system and included the conditions into each pre- and postcondition. To get the same starting point for all strategies, we used this transformation for all experiments. As illustration of the application of the strategy *hidden configuration*, we assume that a method *meth* with a precondition ϕ and a postcondition ψ in our metaproduct exists and, for simplicity, that feature f is not part of both conditions. Before we can apply the strategy *hidden configuration*, we include the invariant into the pre- and postcondition. The result is $\phi \wedge (f \rightarrow inv)$ and $\psi \wedge (f \rightarrow inv)$. Afterwards, we can apply the strategy *hidden configuration*, which leads to $\phi \wedge inv$ as pre- and ψ as postcondition. As the result of the simplification, the invariant is included in the pre- but not in the postcondition and, thus, verification problems can occur. For instance, if we have two method calls to the same method, it is possible that the precondition for the second call is violated since the postcondition of the first call does not ensure the condition of the invariant. Even if this was no problem in our experiment, a possible solution for this problem is to add the feature f to the interface. However, this would increase our interfaces and hinders evolution.

Framing Conditions

Our experiment is based on method contracts that describe under which condition we can call a method and what the method ensures as a result. In addition, it is also common to use framing conditions to specify which fields a method is allowed to change. However, if we want to remove a feature to create one of our interfaces, it can happen that we also have to remove a field that is used in a framing condition. How often this case occurs depends on the strategy that we use for our variable interface (cf. Section 5.3). Thus, if the variable interface presents only members that were introduced in the features of the feature-model interface (cf. simple filtering) this case is more likely. Using the advanced filtering strategy, the probability to remove such a field reference is decreased. Nevertheless, it is possible that a situation occurs in which we have to remove a field and we are not sure about the consequences for the verification. In future, we have to analyze whether problems can occur. However, a similar scenario also exists in single systems. For instance, it is possible that a subclass changes an already existing framing condition. In this case, data groups are used [Leino 1998]. It is an open question, whether data groups are also applicable in our scenario.

Alternative Values

In our experiment, we focused on method contracts. Besides the mentioned problems and limitations in this area, we also identified problems with alternative field definitions. For instance, the feature *DailyLimit* introduces a field `DAILY_LIMIT` to store the limit of one day that we can withdraw. The field is initialized with `-1000`. Now we assume to introduce another feature *ExtendedDailyLimit* that overwrites this daily limit so that the field is initialized with `-2000`. As a result, the initialization of the field in our metaproduct depends on the selection of feature *ExtendedDailyLimit*. Therefore, it is not possible to remove the feature *ExtendedDailyLimit*. One solution of this problem is a transformation of each field access to an access using a getter method. We know that similar problems exist during the verification of alternative types in product lines [Kästner et al. 2012a; Thüm et al. 2014]. In this scenario, a field is typically duplicated and renamed so that two unique fields exist [von Rhein et al. 2016]. In future work, we have to investigate whether this strategy is also a solution for the problem of alternative field initializations in our use case.

6.4 Summary

In this chapter, we introduced the behavioral product-line interface. The behavioral product-line interface is a behavior agreement of interacting elements of two product lines and it is based on all upper-level interfaces, i.e., the feature-model interface and the variable interface. Similar to the upper-level interfaces, we exemplified our behavioral product-line interface for product lines based on FOP written in Java. Therefore, we used JML and variability encryption for the product line and the interface specification. Similar to the interfaces of the other levels, the idea of the behavioral product-line interface is to hide unnecessary details (i.e., specification artifacts of hidden features) of a product line that we plan to reuse.

We used the behavioral product-line interface as proof of concept to investigate whether our concept of multi-level interfaces can be applied to further levels as modeling and implementation. Therefore, we introduced multiple strategies to achieve a behavioral product-line interface. In detail, the strategies *true* and *false configuration* bind the variability of all features that we plan to hide. In contrast to these strategies, the strategy *hidden configuration* hides variability by considering both cases in which a feature is considered as `true` or `false`. The procedure is similar to the generation strategy of the feature-model interface.

As all of the introduced strategies yield advantages and drawbacks, we investigated the practicability of all three strategies. According to Hypothesis 3, we examined whether each strategy can be used to improve the performance of verifications in multi product lines. As a result, we found that all strategies can reduce the verification effort, whereas the strategy *hidden configuration* presents the best results. However, we assumed that it is possible to create an interface automatically, so that it is not necessary to reverify the interface against the implementation. It is part of future work to prove that the

interfaces are correct by construction. Furthermore, based on our subject system, we identified some limitations regarding framing conditions, alternative values, and class invariants so that future work is required.

In sum, we have shown that our concept of multi-level interfaces is applicable to advanced concepts, such as the specification of the product line's behavior. However, we think that it is possible to extend our concept of multi-level interface to further concepts. For instance, we also presented a further idea in the proposal of multi-level interfaces, in which we suggest to apply the concept to non-functional properties [Schröter et al. 2013a]. However, this is an option for future work.

7. Related Work

With the concept of multi-level interfaces, we cover multiple research areas that deal with the modeling, implementation, and with the verification of product lines and multi product lines. As a consequence, this chapter also presents related work to each of these research areas. To ease the overview, the chapter is structured as follows. First, we present concepts that are related to our overall concept of multi-level interfaces, i.e., we present concepts that consider multiple development levels at the same time. Second, we take a look at concepts that are related to our construction mechanisms behind each interface level of our multi-level interfaces. Third, we consider related work that aims to achieve similar goals as our concept.

7.1 Related Work for the Overall Concept

In this section, we present related work for the overall concept of multi product lines that focus on multiple levels of the development. To be more precise, multi product lines as system of interdependent product lines describe a flexible approach that address the problem of combining reusable software artifacts to our needs. In this context, we proposed multi-level interfaces to ease the development process of multi product lines and to improve the encapsulation of the interdependent product lines. However, further concepts with another perspective on the problem of combining variable software artifacts exist, that we present in the following.

As also described in the introduction chapter of this thesis, the efficient development of software systems is based on different concepts regarding the reuse of software artifacts. For instance, software components and modules also represent such reusable software artifacts on different levels of granularity that we can reuse in multiple systems. In this context, we want to highlight the Koala components of van Ommering et al. since the concept also allows the user to combine a system of hierarchical components in a flexible manner [van Ommering et al. 2000]. The concept of Koala component is based

on a definition of requires and provides interfaces that guarantee a correct combination of the involved components. Furthermore, the concept also provides a variability mechanism in which a configuration is used to define the applied connections of alternative subcomponents. If possible, the configuration is used at compile-time and the components are connected based on static binding. Besides the static binding, Koala components also support a dynamic binding that is realized based on switches. However, the interfaces on the implementation level are static whereas the concept of multi product lines allows us to define variable implementation interfaces for which we can use our multi-level interfaces.

In addition, the work of Reiser et al. is a more general view on the management of hierarchical components [Reiser et al. 2009]. By contrast to the Koala components, Reiser et al. use feature models to describe the variability of a components and subcomponents. Thus, the public variability of the component's feature model is mapped to the internal structure such as implementation artifacts but also subcomponents. In this context, Reiser et al. introduce several patterns to handle variability of internal components, such as propagation of the variability to the upper-level component or direct binding of the variability. As a result, the concept also supports variability hiding as the internal variability is not public in all cases. By contrast, in the context of multi-level interfaces, we are not interested in direct binding and we only propagate features that are relevant in the context of the multi product line. Thus, during a concrete instantiation of an underlying product line, some configuration decisions are still open and, depending on the stakeholder, it is possible to configure the underlying product line to the given needs.

A concept for the development of multi product lines that considers the modeling as well as the implementation of underlying product lines was proposed by Damiani et al. for the composition-based approach Delta-Oriented Programming (DOP) [Damiani et al. 2014]. The concept is an extension of DeltaJ to support multi product lines that was designed for product line written in Java for the paradigm of DOP. Similar to a product line based on DOP, a multi product line consists of a code base and a declaration that was extended for the needs of dependent product lines. For instance, the declaration was extended by an import mechanism that enables the user to import another product line, (de)select specific features or add further cross-tree constraints. On the implementation level, for instance, it is possible to rename classes so that the reused product line is compatible to the context of the multi product line. By contrast to this extension of DeltaJ in which the dependent product lines are closely coupled, we focused on the avoidance of direct dependencies between the different product lines of the multi product line. Furthermore, even if we used FOP for our description and evaluation, we focused on a general concept that we can use for multiple implementation techniques from composition as well as annotation-based approaches.

Another related concept to our multi-level interfaces for multi product lines is the variability-aware module system. In detail, the variability-aware module system of Kästner et al. also focuses on modules that can be type checked in isolation but in addition, the systems allows variable interfaces in between [Kästner et al. 2012b]. Thus, the

concept of Kästner et al. allows us to define variability inside of modules and on its interfaces and at the same time it ensures that the well-typedness of modules holds during its composition. Therefore, the structure of this module system is similar to our focused concept of multi product lines since the module is comparable to one product line and the whole system to multi product lines. Furthermore, the variability-aware module system not only considers the implementation of modules but also configuration options. However, by contrast to the described variability of these module systems in which the complete variability of the modules is represented in their interfaces, our concept of multi-level interfaces only focuses on artifacts of interest. In detail, feature-model interfaces additionally hide variability that is not of our interest and we use this information on the implementation level to also hide implementation artifacts that are subsequently not applicable. In addition, if we also consider the extended concept of the variable interface with our feature-context interfaces, we also provide an overview of all members that are safely accessible in a specific implementation context. Last but not least, with our concept of multi-level interfaces, we also focused on a support of further development levels of product lines, such as the behavior level using our behavioral product-line interfaces.

7.2 Related Work for the Interface Construction

In this section, we present related work regarding the construction mechanism of the different interface levels. Therefore, we structure the section according to our multi-level interfaces.

Related Work for the Construction of Feature-Model Interfaces and Feature-Model Compositions

In this section, we consider related work for the concept of feature-model interfaces. As our proofs regarding the analysis results of automated analyses for feature models are based on a combination of feature-model interfaces and feature-model composition, we also consider related work for the composition of feature models.

As already mentioned in Chapter 4, for the construction of feature-model interfaces, we can use the concept of feature-model slicing and the algorithm to remove abstract features from a feature model. In detail, Thüm et al. define an abstract feature as a special kind of feature that can be used to structure other features of the feature model without an impact on the implementation level [Thüm et al. 2011a]. As result, product-line configurations that only differ in the selection of abstract features represent the same implementation and, thus, the same products. To determine the set of product line's products with different implementations, Thüm et al. present an algorithm to remove abstract features from the logical representation of feature models. With this feature-model adaption it is possible to use the state-of-the-art analysis *number of products* to determine the set of different products. By contrast, the algorithm of feature-model slicing is similar but it completely focuses on another application. In detail,

Acher et al. proposed the slice operator to investigate the potential regarding feature-model decomposition [Acher et al. 2011]. The slice operator takes a feature model as input to create a new feature model with unchanged feature dependencies that only consists of features of interest [Acher et al. 2011]. With function S, we used a similar definition as given by the slice operator to create our feature-model interfaces. Based on this function, we presented our proofs regarding analysis-result relation between feature-model compositions with and without these interfaces.

In theory, we can use both algorithms of Acher et al. and Thüm et al. to create our feature-model interfaces. Nevertheless, because of problems regarding the scalability when creating feature-model interfaces, we designed a new algorithm. Krieter et al. present details of our new algorithm and investigate advantages and drawbacks of underlying concepts [Krieter et al. 2016a,b]. In addition, the paper of Krieter et al. also presents an extended evaluation to get insights into appropriate application scenarios of each concept.

As feature-model composition is a central part to describe multi product lines, we also needed a clear definition of it to present our proofs regarding the dependencies of analysis results. In general, the mechanism of feature-model compositions are often described in connection with modeling languages for multi product lines, such as Familiar [Acher et al. 2013a], Velvet [Rosenmüller et al. 2011; Schröter et al. 2013b], TVL [Classen et al. 2011], and VSL [Abele et al. 2010]. In this context, Eichelberger and Schmid present an overview of modeling languages for multi product lines including a comparison regarding their support for composition, modularity, and evolution [Eichelberger and Schmid 2013].

By contrast to this general view on the facilities of modeling languages, Acher et al. investigate different composition operators including their advantages and drawbacks of possible implementations [Acher et al. 2013b]. According to the purpose of a stakeholder, multiple strategies of composition operators exist to combine the configuration of similar feature models. For instance, one option is to create a feature model that represents the union of the configurations of the input feature models, another option is to represent the intersection [Acher et al. 2013b]. However, the paper mainly focuses on a composition of similar feature models using merge strategies, whereas the described composition of this thesis mainly focuses on a combination of different models. In addition to the mentioned paper, the thesis of Acher identifies three different kinds of feature-model compositions, *insert*, *aggregate*, and *merge* that were integrated in Familiar [Acher 2011; Acher et al. 2013a]. The operator aggregate allows the user to combine feature models below a synthetic root and to add new cross-tree constraint between them, whereas the insert operator inserts a complete feature model below a feature of another feature model using a user-defined connection. By contrast, our investigations regarding the feature-model composition and our resulting definition were influenced by the composition introduced with the modeling language Velvet [Rosenmüller et al. 2011]. If we compare our definition of the feature-model composition with the operators of Familiar, we can consider it as an operator that somehow combines the facilities of Familiar's operators aggregate and merge.

Related Work for the Construction of the Variable Interface and Feature-Context Interfaces

In this section, we take a look into related work regarding techniques that we used to create the variable interface and the feature-context interfaces.

Even if the ideas of a syntactical product-line interface are more general, we mainly focused on a concrete implementation of this concept. In detail, we proposed the variable interface as syntactical interface for the implementation of product lines that are based on FOP and considered how to create feature-context interfaces to identify safely accessible API members. To be more precise, our implementation and evaluation considered FOP product lines based on the composer FeatureHouse [Apel et al. 2009, 2013b]. In detail, to collect the API members of the variable interface, we used Fuji as corresponding type checker [Apel et al. 2012; Kolesnikov et al. 2013] for product lines written with FeatureHouse. If we want to use our concept for other composers, languages or implementation types, two options exist. We can write an own mechanism to collect the necessary data or, if available, we need to extract the data from a corresponding type checker. However, if a type checker for our purposes exists, depends on the kind of product-line implementation (i.e., annotation-based or composition-based) as well as on the concrete realization (e.g., the composer) and on the supported language. For instance, the type checker Fuji is a type checker for the composition-based implementation technique of FOP and supports product-lines written in Java [Apel et al. 2012; Kolesnikov et al. 2013]. Even if the composer FeatureHouse supports further languages for FOP, the type checker Fuji is limited to the language Java. Nevertheless, some other composition-based implementation techniques exist that also provide approaches for the efficient type checking that we could use to extend our support for feature-context interfaces. For instance, Schaefer et al. present an approach for the efficient type checking of product lines based on DOP [Schaefer et al. 2011]. Besides the composition-based product lines, it is also possible to extend our approach to annotation-based implementation techniques. In this context, we can use a corresponding type checker to create an extension of our feature-context interfaces. For this purpose, we could use the type checker introduced by Kästner et al. that supports type checks for annotation-based product lines written in Java using preprocessors [Kästner et al. 2012a].

Related Work for the Construction of the Behavioral Interfaces

To complete the overview of related work regarding the construction of multi-level interfaces, we also briefly describe the underlying mechanisms that can be used to create a behavioral product-line interface.

The thesis presents three strategies that we can use to construct a behavioral product-line interface. Whereas the strategy *false* and strategy *true configuration* are straight forward, the creation process for the strategy *hidden configuration* is more complex. However, the construction process of this strategy is based on the same mechanisms as we presented for the feature-model interface. In detail, it is also possible to use

the mechanisms to remove abstract features [Thüm et al. 2011a] or the concept of feature-model slicing [Acher et al. 2011] to create this kind of a behavioral product-line interface. For more details, we refer the reader to the corresponding paragraph in the beginning of this section.

7.3 Related Work Considering Similar Goals

As our concept of multi-level interfaces also aims to support multiple properties, we use this section to present related work that also aims to achieve these goals. In detail, we take a look at concepts for information hiding and evolution and consider product-line analyses.

7.3.1 Information Hiding and Evolution

Information hiding is one the main goals that we addressed with our concept of multi-level interfaces. However, for each of the considered levels of the product-line development exists related work that directly or indirectly focuses on this property. Considering the modeling level of product lines, several views were proposed to ease the configuration process for the developer [Hubaux et al. 2010; Mannion et al. 2009; Schroeter et al. 2012]. In general, the views present an excerpt of an underlying feature model that is customized to the needs of the developer, so that the decision process during a configuration is simplified. In contrast to these view concepts, the concept of feature-model interfaces does not only hide variability of one underlying model; we can use the feature-model interface as a representation of multiple compatible feature models. Furthermore, Dhungana et al. propose the concept of model fragments that represents parts of variability models, which can be merged to one system [Dhungana et al. 2010]. In this context, some of the internals of model fragments are public whereas other are private and, thus, hidden to other model fragments. In contrast to our concept of feature-model interfaces, the concept mainly considers consistency checks between the models and the represented system, whereas we focused on an investigation of automated analysis of the composed models especially for feature models.

Considering the part of product-line implementation, several techniques exist to support information hiding. For instance, Kästner et al. propose the *Colored Integrated Development Environment (CIDE)* that was developed to support the implementation of annotation based product lines [Kästner et al. 2008a]. For the purpose of advanced implementation support, CIDE also provides multiple views that present a subset of all implementation units [Kästner et al. 2008b]. In detail, Kästner et al. present three views: the *feature view* for feature-specific code, the *variant view* that considers the code for a variant, and the *realization view* that presents code of a specific feature including additional code of dependent code units. However, the *realization view* only considers additional code units that need to be available for a valid code structure, whereas the feature-context interface also considers further code units according to the feature dependencies. In addition, Apel et al. propose access modifier for the paradigm of FOP to customize the access to implementation units [Apel et al. 2012]. In this context, the

authors introduce the modifiers, *feature*, *subsequent*, and *program*. Although the concept allows us to hide implementation units, it does not focus on a view that presents safely accessible members. By contrast, this concept presents access modifiers similar to access modifiers of programming languages to encapsulate functionalities. Thus, the concept of feature-context interfaces and the access modifiers are orthogonal. Therefore, it is possible to tailor the safely accessible API members of the feature-context interface using these access modifiers.

Besides the mentioned aims of multi-level interface for information hiding, our concept also focuses on a support for evolution. To investigate the evolution of plugin-based systems, Acher et al. presented a process to extract and compare different versions of feature models from plugin-based systems [Acher et al. 2014]. For this purpose, Acher et al. use the slice and the aggregate operator of Familiar [Acher et al. 2013a] to extract and create feature models for the different versions of an investigated plug-in system. A subsequent comparison of these feature models allows a user to draw conclusions about the impact of evolutionary changes. In addition, as already mentioned in the Section 7.3.1, the concept of model fragments of Dhungana et al. also focuses on the support of evolution [Dhungana et al. 2010]. In detail, the authors create a merge history so that the maintenance of the system and the process for a fragment's remerge can benefit from information of previous versions. In addition to the mentioned approaches, Vierhauser et al. propose an approach to check the consistency of evolving variability models including the consistency to the code base [Vierhauser et al. 2010]. The approach focuses on the support of step-wise changes regarding the variability-model dependencies, in which the user is interested in immediate feedback regarding inconsistencies. By contrast to these concepts, our investigation of multi-level interfaces with the concept of feature-model interfaces and feature-model composition focuses on a product-line evolution that prevents a user from reanalyzing the whole multi product line. Thus, our main focus is to stabilize the analysis process so that an evolved product line does not necessary lead to changes in the whole multi product line.

7.3.2 Product-Line Analyses

As the concept of multi-level interfaces simplifies multiple analyses on different levels of the product-line development, this section considers related work according to each level. To classify the approaches, we start this section with a brief overview of general concepts on product-line analyses. Afterwards, we consider related work for each level of our multi-level interfaces.

General Concepts to Analyze Product Lines

In general, multiple approaches were proposed to scale the product-line analyses that can be applied to different levels of the product-line development. To get an overview, Thüm et al. summarize these approaches to general concepts and present an overview of representative approaches [Thüm et al. 2014a]. In theory, three main analysis concepts with different advantages and disadvantages exist, *feature-based*, *product-based*, and

family-based. Whereas *feature-based* and *product-based* concepts focus on the analysis of each feature or rather each product in isolation, the *family-based* approach focuses on the analyses of the whole product line. For this purpose, the *family-based* concepts use the feature dependencies as input for the specific analysis approach. However, as mentioned, these general concepts can be used on different levels. For instance, considering the analyses of the implementation level, Kolesnikov et al. compares the usage of each concept regarding the area of type checking for product lines written with FOP [Kolesnikov et al. 2013].

Analyses for the Modeling Level

As Kang et al. proposed feature models, they also mentioned that it is necessary to analyze the described feature dependencies to ensure the feature-model correctness [Kang et al. 1990]. In this context, the authors also present first feature-model analyses, such as the analysis of void feature models. Since this starting point, several authors proposed additional analyses to avoid anomalies or to create statistics. Benavides et al. present an overview of existing feature-model analyses and refer to implementations and tool support [Benavides et al. 2010]. We used some of these analyses for the evaluation of conceptual parts of multi-level interfaces and described analysis dependencies between feature-model compositions with and without feature-model interfaces.

In the context of tool support for feature-model analysis, Mendonça et al. clarified that, even if the feature-model analysis represents a np-complete problem, the analyses scales well for the domain of feature models [Mendonça et al. 2009b]. Thus, the tool support for feature-model analyses and configuration is in general based on satisfiability solvers or binary decision diagrams [Acher et al. 2013a; Benavides et al. 2007; Mendonça et al. 2009a; Thüm et al. 2014b]. Tools like FeatureIDE [Thüm et al. 2014b] allow us to identify anomalies or to collect statistics for the feature models. However, even if a satisfiability check for a feature model, in general, presents results in an appropriate time, the concrete complexity depends on the specific analysis (cf. the analysis of atomic sets) and on the complexity of the feature model. In this context, feature-model interfaces can help us to focus on the features of interest for feature-model compositions and are able to prevent us from reanalyzing feature-model compositions in some cases. However, the concept of feature-model interfaces does not change the underlying analyses so that the concepts can be integrated in the existing tool support.

As already described in the background chapter, besides feature modeling, other concepts for variability modeling exist, such as decision modeling [Schmid et al. 2011], and orthogonal variability modeling [Pohl et al. 2005]. In the context of ecosystems, Galindo et al. propose a concept to configure multiple variability models based on different variability modeling approaches [Galindo et al. 2015]. For this purpose, the authors present insights on how to extend the tool invar to further variability models and how to describe and check their inter-model constraints. Therefore the different variability types are mapped to invar types. However, we assume that this kind of a mapping between variability models can also be used in our approach to support product lines based on other variability-model approaches.

Analyses for the Implementation Level

Based on type checking during the compilation of a program it is possible to detect errors in early states of the product implementation [Pierce 2002]. In general, it is also possible to use existing type checker to investigate product line's products. This means, we are able to create a specific product and to apply a type-checker on it. As described above, this is a product-based approach that does not scale for thousands of possible products. By contrast, family-based type checkers focus on a type check for the whole product line so that a subsequent check of a product line's product is superfluous. In this context, Kästner et al. presents an efficient approach for family-based type checking of annotation-based product lines to type check a corresponding product line as whole [Kästner et al. 2012a]. In a similar manner, efficient approaches exist for the type checking of product lines implemented based on FOP [Apel et al. 2010; Kolesnikov et al. 2013], and DOP [Schaefer et al. 2011]. In addition to these concepts, Kästner et al. present a variability-aware module system that allows the developer to implement variable modules with variable interfaces in between [Kästner et al. 2012b]. The concept of Kästner et al. also supports type checks of each module in isolation. Thus, it is not necessary to combine the modules for the purpose of type checking. By contrast to these concepts, we introduced the variable interface as part of our multi-level interfaces and combined it with the concept of feature-context interfaces so that we are able to detect errors a priori. Thus, the concept of the variable interface with feature-context interfaces is able to support the developer during the implementation task so that it is possible to prevent the developer from type conflicts.

Another concept to support the implementation task of product lines was presented by Ribeiro et al. with emergent interfaces [Ribeiro et al. 2010]. In detail, an emergent interface focuses on the maintenance of annotation-based product lines and presents information of and for other features by presenting artifacts that are provided and required. The concept is based on dataflow analyses and it, similar to our feature-context interfaces, helps to ease the error-prone development step of product lines. Based on the concept of emergent interfaces, Thüm et al. present an extended version of these interfaces with additional contract information [Thüm et al. 2016]. For instance, based on existing method contracts, it is possible to extract the range in which a value of a method parameter is valid. This information is used to extend the emergent interfaces to emergent contract interfaces. As a result, if such an information is available, it is not necessary for the developer to extract this important information manually so that time and effort can be saved. However, as mentioned above, the concepts mainly focus on annotation-based product lines, whereas our concept was introduced for the concept of FOP. Nevertheless, in future work, it is possible to generalize our concept to other composition-based approaches or implementation paradigms.

Analyses for the Behavioral Level

As mentioned above, the analysis of product lines can be associated to family-based, product-based, and feature-based strategies as well as combinations thereof [Thüm et al.

2014a]. In particular, if we consider analyses for the behavioral level, the most concepts are based on these strategies. Therefore, we briefly consider proposed concepts and structure the following paragraphs accordingly.

First, we consider optimized product-based strategies. In this context, Bruns et al. proposed a delta-oriented slicing that also aims to reduce the effort for deductive verification [Bruns et al. 2011]. The concept of DOP uses deltas to describe the difference between two products. In this context, the algorithm of delta-oriented slicing identifies the proofs that are affected by the deltas of two input products. Using the resulting information as input, the concept of Bruns et al. focuses on a reuse of existing parts of proofs from the first product so that it is only necessary to partly reanalyze the second product. In addition to this approach, Hähnle et al. proposed abstract method calls to also improve the product-based strategy [Hähnle et al. 2013]. The concept was introduced for DOP and it also focuses on a reuse of already existing proofs. In detail, the concept introduces abstract contracts to explicitly separate the contract in a reusable abstract and in a concrete part.

By contrast to the product-based strategies, Damiani et al. proposed a feature-product-based strategy for the purpose of product-line verification [Damiani et al. 2012]. In detail, a feature-product-based strategy is a mixture of the feature-based and a product-based strategy. This means, some properties are checked features-based, whereas other properties needs to be checked using the product-based strategy as a feature-based analysis is not sufficient [Thüm et al. 2014a]. Considering the concept of Damiani et al., this strategy can be transformed as follows. First, the deltas are verified in isolation based on symbolic assumptions. Afterwards, during the product generation, these assumptions are replaced by instances so that already verified aspects are efficiently reused in the context of the product verification. Another feature-product-based approach was proposed by Thüm et al. based on proof compositions [Thüm et al. 2011b]. Here, the idea is to use composed Coq proof scripts of features (i.e., partial proofs) to verify the individual products.

Another mixture of the general analyses strategies is the feature-family-based strategy. In this context, the feature-based strategy is applied first, followed by the application of a family-based strategy for all artifacts that we cannot analyze in isolation. One representative of this strategy in the context of behavior analyses, is the concept of Hähnle and Schaefer. The authors proposed a form of the Liskov principle that can be applied to DOP [Hähnle and Schaefer 2012]. In detail, the Liskov principle can be used to modularize the verification in the context of OOP based on inheritance relations [Liskov and Wing 1994]. For the purpose of applying the concept to DOP [Hähnle and Schaefer 2012], the authors propose to adapt the concept to verify the core and each delta of the DOP product line in isolation. Afterwards, it should be possible to conclude to verification results for the complete product line.

In sum, if we consider the described analysis strategies, multi product lines are not explicitly focused. Therefore, there is no evaluation regarding multi product lines and, thus, it is not clear to which extend the approaches are applicable in this area. Furthermore, most of these strategies for the analysis of the behavioral level are based on the generated products. As a result, the concepts are not applicable to huge product lines as the generation and verification of all products is almost infeasible.

8. Conclusion and Future Work

The reuse of software artifacts is a central part of the software-development process. Especially when we focus on the development of similar programs tailored to the needs of different customers, the reuse of software artifacts is inevitable to achieve an efficient development process. In this context, software product lines provide mechanisms to develop similar programs from a common code base and, thus, to improve the reuse between products. Nevertheless, even if software product lines are well-established, the growing complexity of current systems results in new challenges, for instance, to ensure a correct development or to analyze the whole system. To overcome these challenges, the concept of multi product lines, which describes a set of interdependent product lines, provides new techniques to lift the reuse to another level. As a result, besides the reuse of individual software artifacts, we can also reuse entire product lines. However, direct dependencies between the product lines of a multi product line exist that can hinder the distributed development and extension of the underlying product line. For instance, if one product line of a multi product line is changed (e.g., the modeling or implementation), it needs to be checked whether these changes have an impact on other product lines that also leads to a necessary adaption.

To avoid direct dependencies between the product lines of a multi product line and to ease product-line analyses and evolution, we proposed the concept of multi-level interfaces. Thus, the thesis focused on the presentation of the holistic concept of multi-level interfaces with its underlying interface levels that depend on each other. In detail, we introduced and refined multiple interfaces to avoid direct dependencies during the modeling, the implementation, and the specification of the behavior between the involved product lines. For each level, we also defined one hypothesis for each interface that we used in our evaluations to investigate our global research questions:

> **GRQ1:** How can we represent the interfaces of the development levels of a multi product line?

GRQ2: Is it possible to generate the interfaces of the development levels of a multi product line?
GRQ3: Can we use multi-level interfaces to improve the analysis and/or evolution of multi product lines?

In the following, we consider the answers of our global research questions and present our contributions:

1. We defined feature-model interfaces to avoid the direct dependencies between the involved product lines on the modeling level of multi product lines. The feature-model interface is itself a feature model that only consists of the features that are relevant in the context of the multi product line (cf. GRQ1). Based on this definition, we proved how to profit from this interface during the analysis of composed feature models in the context of multi product lines. We found that concepts exist that allows us to generate our feature-model interface (cf. GRQ2). However, because of scalability problems we designed a new algorithm for our evaluation even if the algorithm itself was out of scope of this thesis. Based on this, we investigated research question GRQ3 regarding the modeling level. In detail, we evaluated the potential of feature-model interface in combination with feature-model composition to ease the automated analysis using a three-month snapshot of a real-world product line. As one result, feature-model interfaces are able to prevent the re-execution of automated analysis during evolution in more than half of all considered cases.

2. We defined the variable interface as an interface to encapsulate the variability on the implementation level of the involved product lines. In detail, we used a list of all product line's API members with additional variability information (i.e., using presence conditions) that we filtered to the features of interest according to the upper modeling level (cf. GRQ1). We also showed how to automatically generate this interface, as a manual creation is not suitable (cf. GRQ2). In addition, based on the concept of a variable interface, we proposed the concept of feature-context interfaces as a non-variable view on reusable API members. To investigate our global research question GRQ3, we integrated tool support into FeatureIDE and compared our feature-context interfaces with state-of-the-art approaches to identify accessible API members in real-world product lines. Our findings show that only feature-context interfaces provide an overview of all safely accessible API members whereas all other approaches present unsound or incomplete results. Thus, the variable interface in combination with the feature-context interfaces is able to ease the implementation of (multi) product lines so that a developer does not need to manually analyze existing feature and code dependencies.

3. As proof for the extensibility of our holistic concept of multi-level interfaces, we also considered the behavioral level of the product-line development. In detail, based on feature-oriented contracts, we presented three strategies to encapsulate the specification of the different product lines of a multi product line (cf. GRQ1).

Using a small example of a multi product line that is based on two product lines, we investigated the application of each strategy to answer research question GRQ3. In this context, we defined the feature-oriented contracts with FeatureIDE and applied the program verification with KeY. As a result, we found that the strategies are able to reduce the verification effort. However, even if we focused on concepts that allow us to automatically generate the interfaces in future (cf. GRQ2), we created the interfaces manually. Thus, it is an open question whether we can ensure that the interfaces are correct by construction and, thus, further research in this direction is necessary. Nevertheless, this investigation of concepts for the behavioral product-line interface approves the extensibility of multi-level interfaces.

4. We illustrated the holistic concept of multi-level interfaces showing the relationship of each interface level to the corresponding upper levels. This illustration also relates to research question GRQ1 as it was essential to find interface representations that are able to consider the relations to the upper interface levels. However, with our feature-model interface, the variable interface, and the strategies of the behavioral product-line interface, we found a suitable representation for each interface level that also considers these relations. Even if we created the behavioral product-line interface manually, we already found solutions for the generation of the feature-model interface and the variable interface. Thus, we can answer the research question GRQ2 partially. In addition, we also presented four cases of evolution for multi product lines when using multi-level interfaces and gave an overview of the effects on the whole system. Based on this, we can also answer the research question GRQ3 regarding evolution in a general scope. We found that only in one of the cases, in which changes in a reused product line are not compatible to the specific interface, we are forced to reanalyze the level of the multi product line.

In sum, we conclude that multi-level interfaces help to avoid direct dependencies between product lines of a multi product line. As a result, it is possible to use a specific interface of our multi-level interface to ease corresponding analyses and evolution. For instance, using feature-model interfaces, we can reduce the complexity of feature-model compositions. Here, we proved that he automated analysis based on feature-model compositions with and without a feature-model interface depends on each other so that we can conclude from one analysis result to the other. In this context, we also showed that feature-model interfaces can be used to ease the analyses during feature model evolution as it can prevent us from reanalyzing the complete feature-model composition (i.e., if the new feature model is also compatible to the feature-model interface). In addition, using feature-context interfaces, we can reduce the effort for the manual analysis of a product-line, in which the developer searches for reusable implementation artifacts. In this context, feature-context interfaces are able to present safely accessible API members and, thus, prevent developers from type errors and optimize the efficient source-code reuse. To complement our investigation, we used the behavioral

product-line interface as a proof of concept for the extensibility of multi-level interfaces, in which we also identified the potential for the analysis process of the correct behavior of product-line's products. Therefore, the concept of multi-level interface complies with our initial goals to avoid direct dependencies between product lines and to ease the analyses and evolution of the whole system of multi product lines.

Future Work

We identified several options for future work. First of all, we consider future work for the presented interfaces and give a brief outlook on possible extensions. Second, we present further concepts for future work including an extension for our holistic concept of multi-level interfaces.

As one interface of our concept of multi-level interfaces, we proposed feature-model interfaces. In this context, we presented several proofs regarding the analysis results of feature-model compositions with and without feature-model interfaces according to a set of state-of-the-art feature-model analyses. In this thesis, we mainly focused on automated analyses that are most popular in the community. However, Benavides et al. summarized further analyses [Benavides et al. 2010] so that it is possible to extend our proofs by some of these analyses in future work. In addition, we investigated feature-model interfaces by analyzing a real-world multi product line. Unfortunately, we had only access to four snapshots of a very early state of this multi product line so that in future work an extended evaluation would be desirable.

Regarding the variable interface and the feature-context interfaces, we illustrated that we can use feature-context interfaces to identify safely accessible API members for the development of product lines and multi product lines. We evaluated the concept on an ordinary product line to illustrate its advantages. We also applied a user study to compare the state-of-the-art mechanisms with our feature-context interfaces for the task of identifying safely accessible API members. Unfortunately, the small number of participants does not allow us to yield usable conclusions. Thus, in future work it is a desired option to reapply the user study in order to gain additional insights into the real potential of the variable interface with feature-context interfaces. Furthermore, our current tool integration for feature-context interfaces in FeatureIDE only supports the development of product lines but not multi product lines. Therefore, tool support for multi product lines is still an open task for future work.

As a proof of concept for our multi-level interfaces, we also considered the behavioral level regarding the development of multi product lines. In this context, we proposed three strategies to realize the behavioral product-line interface. For our evaluation, we created hand-written behavioral product-line interfaces so that it was possible for us to find an appropriate strategy. Even if the strategy of a *hidden configuration* presented the best results for a behavioral product-line interface, the task to automatically generate the interface is still open. In future work, it is possible to extend our tool support to provide mechanism for the generation of the behavioral product-line interface. Furthermore, to ensure the correctness of the interfaces for the evaluation, we additionally

verified each interface. This step is not necessarily needed, if it is possible to prove that the interface is correct by construction. However, such a proof is an open task for future work, in which it needs to be ensured that an interface generation based on a verified product line does not break existing proofs.

Besides the open tasks for future work regarding each presented level of the multi-level interfaces, it is also possible to extend the holistic concept by new interface levels. For instance, the originally proposed idea of multi-level interfaces also includes an interface for non-functional properties [Schröter et al. 2013a]. In detail, non-functional properties, such as performance or footprint, can be affected when we (de)select a specific feature [Siegmund et al. 2011]. The estimation of a feature dependency to a non-functional property could be a complex task because we also need detailed knowledge about the feature interactions of the underlying product lines. An interface for this level, in which these properties are described, can be used to encapsulate these estimations for the whole multi product line. In future work, someone can refine these ideas so that it is possible to develop further tool support based on it. Furthermore, the current tool support for the proposed levels of modeling and implementation is tailored to the needs of our evaluations (e.g., the feature-context interface is tailored to a scenario of single product lines). Thus, an improved and extended implementations is needed to ease the development of multi product lines in practice and to enable further advanced evaluations based on multiple complex multi product lines.

Bibliography

Abele, A., Papadopoulos, Y., Servat, D., Törngren, M., and Weber, M. (2010). The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 101–105, Germany. Universität Duisburg-Essen.

Acher, M. (2011). *Managing Multiple Feature Models: Foundations, Language, and Applications*. PhD thesis, University of Nice Sophia Antipolis, France.

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2014). Extraction and Evolution of Architectural Variability Models in Plugin-Based Systems. *Software and System Modeling (SoSyM)*, 13(4):1367–1394.

Acher, M., Collet, P., Lahire, P., and France, R. B. (2010). Comparing Approaches to Implement Feature Model Composition. In *Proc. Europ. Conf. Modelling Foundations and Applications (ECMFA)*, pages 3–19, Berlin, Heidelberg. Springer.

Acher, M., Collet, P., Lahire, P., and France, R. B. (2011). Slicing Feature Models. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 424–427, Washington, DC, USA. IEEE.

Acher, M., Collet, P., Lahire, P., and France, R. B. (2013a). FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)*, 78(6):657–681.

Acher, M., Combemale, B., Collet, P., Barais, O., Lahire, P., and France, R. B. (2013b). Composing Your Compositions of Variability Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 352–369. Springer.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg.

Apel, S., Kästner, C., Größlinger, A., and Lengauer, C. (2010). Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300.

Apel, S., Kästner, C., and Lengauer, C. (2009). FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231, Washington, DC, USA. IEEE.

Apel, S., Kästner, C., and Lengauer, C. (2013b). Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79.

Apel, S., Kolesnikov, S., Liebig, J., Kästner, C., Kuhlemann, M., and Leich, T. (2012). Access Control in Feature-Oriented Programming. *Science of Computer Programming (SCP)*, 77(3):174–187.

Apel, S., von Rhein, A., Thüm, T., and Kästner, C. (2013c). Feature-Interaction Detection Based on Feature-Based Specifications. *Computer Networks*, 57(12):2399–2409.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg. Springer.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371.

Beckert, B., Hähnle, R., and Schmitt, P. (2007). *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Heidelberg.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708.

Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2007). FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134, Limerick, Ireland. Technical Report 2007-01, Lero.

Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., and Wąsowski, A. (2013). A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8, New York, NY, USA. ACM.

Bosch, J. (2009). From Software Product Lines to Software Ecosystems. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 111–119, Pittsburgh, PA, USA. Software Engineering Institute.

Bosch, J. and Bosch-Sijtsema, P. (2010). From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems. *Journal of Systems and Software (JSS)*, 83(1):67–76.

Bošković, M., Mussbacher, G., Bagheri, E., Amyot, D., Gašević, D., and Hatala, M. (2011). Aspect-Oriented Feature Models. In *Proc. Int'l Conf. Models in Software Engineering (MODELSWARD)*, pages 110–124, Berlin, Heidelberg. Springer.

Bruns, D., Klebanov, V., and Schaefer, I. (2011). Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 61–75, Berlin, Heidelberg. Springer.

Chen, L. and Ali Babar, M. (2011). A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *J. Information and Software Technology (IST)*, 53(4):344–362.

Classen, A., Boucher, Q., and Heymans, P. (2011). A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming (SCP)*, 76(12):1130–1143.

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010). Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, NY, USA. ACM.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wasowski, A. (2012). Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 173–182, New York, NY, USA. ACM.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005). Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169.

Czarnecki, K. and Pietroszek, K. (2006). Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA. ACM.

Czarnecki, K. and Wąsowski, A. (2007). Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 23–34, Washington, DC, USA. IEEE.

Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., and Yu, I. C. (2012). A Transformational Proof System for Delta-Oriented Programming. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FM-SPLE)*, pages 53–60, New York, NY, USA. ACM.

Damiani, F., Schaefer, I., and Winkelmann, T. (2014). Delta-Oriented Multi Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 232–236, New York, NY, USA. ACM.

Dhungana, D., Grünbacher, P., Rabiser, R., and Neumayer, T. (2010). Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *Journal of Systems and Software (JSS)*, 83(7):1108–1122.

Durán, A., Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2017). FLAME: A Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing. *Software and System Modeling (SoSyM)*, 16(4):1049–1082.

Eichelberger, H. and Schmid, K. (2013). A Systematic Analysis of Textual Variability Modeling Languages. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 12–21, New York, NY, USA. ACM.

Galindo, J. A., Dhungana, D., Rabiser, R., Benavides, D., Botterweck, G., and Grünbacher, P. (2015). Supporting Distributed Product Configuration by Integrating Heterogeneous Variability Modeling Approaches. *J. Information and Software Technology (IST)*, 62(C):78–100.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA.

Hähnle, R. and Schaefer, I. (2012). A Liskov Principle for Delta-Oriented Programming. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 32–46, Berlin, Heidelberg. Springer.

Hähnle, R., Schaefer, I., and Bubel, R. (2013). Reuse in Software Verification by Abstract Method Calls. In *Proc. Int'l Conf. Automated Deduction (CADE)*, pages 300–314, Berlin, Heidelberg. Springer.

Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. (2012). Behavioral Interface Specification Languages. *ACM Computing Surveys*, 44(3):16:1–16:58.

Hemakumar, A. (2008). Finding Contradictions in Feature Models. In *Proc. Int'l Workshop on Analyses of Software Product Lines (ASPL)*, pages 183–190. Lero Int. Science Centre, University of Limerick, Ireland.

Holl, G., Grünbacher, P., and Rabiser, R. (2012). A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *J. Information and Software Technology (IST)*, 54(8):828–852.

Hubaux, A., Heymans, P., Schobbens, P.-Y., and Deridder, D. (2010). Towards Multi-View Feature-Based Configuration. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 106–112, Berlin, Heidelberg. Springer.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute.

Kästner, C., Apel, S., and Kuhlemann, M. (2008a). Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320, New York, NY, USA. ACM.

Kästner, C., Apel, S., and Ostermann, K. (2011). The Road to Feature Modularity? In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8, New York, NY, USA. ACM.

Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012a). Type Checking Annotation-Based Product Lines. *Trans. Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:39.

Kästner, C., Ostermann, K., and Erdweg, S. (2012b). A Variability-Aware Module System. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792, New York, NY, USA. ACM.

Kästner, C., Trujillo, S., and Apel, S. (2008b). Visualizing Software Product Line Variabilities in Source Code. In *Proc. Int'l Workshop Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313. Lero Int. Science Centre, University of Limerick, Ireland.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242, Berlin, Heidelberg. Springer.

Kolesnikov, S., von Rhein, A., Hunsen, C., and Apel, S. (2013). A Comparison of Product-Based, Feature-Based, and Family-Based Type Checking. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 115–124, New York, NY, USA. ACM.

Krieter, S., Schröter, R., Thüm, T., Fenske, W., and Saake, G. (2016a). Comparing Algorithms for Efficient Feature-Model Slicing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, New York, NY, USA. ACM.

Krieter, S., Schröter, R., Thüm, T., and Saake, G. (2016b). An Efficient Algorithm for Feature-Model Slicing. Technical Report 01, School of Computer Science, University of Magdeburg, Germany.

Lauenroth, K., Pohl, K., and Toehning, S. (2009). Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280, Washington, DC, USA. IEEE.

Leino, K. R. M. (1998). Data Groups: Specifying the Modification of Extended State. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 144–153, New York, NY, USA. ACM.

Liskov, B. H. and Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841.

Mannion, M., Savolainen, J., and Asikainen, T. (2009). Viewpoint-Oriented Variability Modeling. In *Proc. Computer Software and Applications Conf. (COMPSAC)*, pages 67–72, Washington, DC, USA. IEEE.

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., and Saake, G. (2017). *Mastering Software Variability with FeatureIDE*. Springer.

Mendonça, M., Branco, M., and Cowan, D. (2009a). S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762, New York, NY, USA. ACM.

Mendonça, M., Wąsowski, A., and Czarnecki, K. (2009b). SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 231–240, Pittsburgh, PA, USA. Software Engineering Institute.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10):40–51.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, USA.

Pleumann, J., Yadan, O., and Wetterberg, E. (2011). Antenna: An Ant-to-End Solution For Wireless Java. Website. Available online at `http://antenna.sourceforge.net/`; visited on March, 2018.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg.

Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443, Berlin, Heidelberg. Springer.

Reiser, M.-O., Kolagari, R. T., and Weber, M. (2009). Compositional Variability - Concepts and Patterns. In *Hawaii International International Conference on Systems Science (HICSS)*, pages 1–10, Los Alamitos, CA, USA. IEEE.

Reiser, M.-O. and Weber, M. (2006). Managing Highly Complex Product Families with Multi-Level Feature Trees. In *Proc. Int'l Conf. Requirements Engineering (RE)*, pages 149–158, Los Alamitos, CA, USA. IEEE.

Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent Feature Modularization. In *Proc. Int'l Conf. Object-Oriented Programming Systems Languages and Applications Companion (SPLASH)*, pages 11–18, New York, NY, USA. ACM.

Rosenmüller, M. and Siegmund, N. (2010). Automating the Configuration of Multi Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, Germany. Universität Duisburg-Essen.

Rosenmüller, M., Siegmund, N., and Kuhlemann, M. (2010). Improving Reuse of Component Families by Generating Component Hierarchies. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 57–64, New York, NY, USA. ACM.

Rosenmüller, M., Siegmund, N., Thüm, T., and Saake, G. (2011). Multi-Dimensional Variability Modeling. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–22, New York, NY, USA. ACM.

Rosenmüller, M., Siegmund, N., ur Rahman, S. S., and Kästner, C. (2008). Modeling Dependent Software Product Lines. In *Proc. Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 13–18. Department of Informatics and Mathematics, University of Passau.

Rubin, J. and Chechik, M. (2013). A Framework for Managing Cloned Product Variants. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1233–1236, Piscataway, NJ, USA. IEEE.

Rubin, J., Czarnecki, K., and Chechik, M. (2013). Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 101–110, New York, NY, USA. ACM.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 77–91, Berlin, Heidelberg. Springer.

Schaefer, I., Bettini, L., and Damian, F. (2011). Compositional Type-Checking for Delta-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56, New York, NY, USA. ACM.

Schink, H., Broneske, D., Schröter, R., and Fenske, W. (2016). A Tree-Based Approach to Support Refactoring in Multi-Language Software Applications. In *International Conference on Advances and Trends in Software Engineering (SOFTENG)*, pages 44–49. IARIA.

Schmid, K., Rabiser, R., and Grünbacher, P. (2011). A Comparison of Decision Modeling Approaches in Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 119–126. ACM.

Schroeter, J., Lochau, M., and Winkelmann, T. (2012). Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 252–268, Berlin, Heidelberg. Springer.

Schröter, R. (2014). Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines. Technical Report 04, School of Computer Science, University of Magdeburg, Germany.

Schröter, R., Krieter, S., Thüm, T., Benduhn, F., and Saake, G. (2016). Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 667–678, New York, NY, USA. ACM.

Schröter, R., Siegmund, N., and Thüm, T. (2013a). Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Software Product Line Conference co-located Workshops*, pages 96–99, New York, NY, USA. ACM.

Schröter, R., Siegmund, N., Thüm, T., and Saake, G. (2014). Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 102–111, New York, NY, USA. ACM.

Schröter, R., Thüm, T., Siegmund, N., and Saake, G. (2013b). Automated Analysis of Dependent Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9:1–9:5, New York, NY, USA. ACM.

Segura, S. (2008). Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Workshop on Analyses of Software Product Lines (ASPL)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland.

Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P. G., Apel, S., and Kolesnikov, S. (2011). Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 160–169, Washington, DC, USA. IEEE.

Tartler, R., Lohmann, D., Dietrich, C., Egger, C., and Sincero, J. (2012). Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14.

Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA. ACM.

Thüm, T. (2015). *Product-Line Specification and Verification with Feature-Oriented Contracts*. PhD thesis, University of Magdeburg, Germany.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014a). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45.

Thüm, T., Batory, D., and Kästner, C. (2009). Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 254–264, Washington, DC, USA. IEEE.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014b). FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85.

Thüm, T., Kästner, C., Erdweg, S., and Siegmund, N. (2011a). Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200, Washington, DC, USA. IEEE.

Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., and Saake, G. (2014). Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 177–186, New York, NY, USA. ACM.

Thüm, T., Ribeiro, M., Schröter, R., Siegmund, J., and Dalton, F. (2016). Product-line Maintenance with Emergent Contract Interfaces. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 134–143, New York, NY, USA. ACM.

Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-Based Deductive Verification of Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA. ACM.

Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. (2011b). Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, DC, USA. IEEE.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. (2012). Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 255–269, Berlin, Heidelberg. Springer.

Thüm, T., Winkelmann, T., Schröter, R., Hentschel, M., and Krüger, S. (2016). Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104, New York, NY, USA. ACM.

Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., and Toro, M. (2008). Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software (JSS)*, 81(6):883–896.

Trinidad, P., Benavides, D., and Ruiz-Cortés, A. (2004). Improving Decision Making in Software Product Lines Product Plan Management. In *Proc. Workshop on Decision Support in Software Engineering (ADIS)*, pages 1–8, Malaga, Spain. RWTH Aachen.

Trinidad, P. and Ruiz-Cortés, A. (2009). Abductive Reasoning and Automated Analysis of Feature Models: How are They Connected? In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 145–153, Essen, Germany. Universität Duisburg-Essen.

van der Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin, Heidelberg.

van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2000). The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85.

Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., and Heider, W. (2010). Flexible and Scalable Consistency Checking on Product Line Variability Models. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 63–72, New York, NY, USA. ACM.

von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., and Apel, S. (2016). Variability Encoding: From Compile-Time to Load-Time Variability. *Journal of Logic and Algebraic Methods in Programming (JLAMP)*, 85(1, Part 2):125–145.

Zhang, W., Zhao, H., and Mei, H. (2004). A Propositional Logic-Based Method for Verification of Feature Models. In *Proc. Int'l Conf. Formal Methods and Software Engineering (ICFEM)*, pages 115–130, Berlin, Heidelberg. Springer.