Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

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

von: Diplom-Informatiker Marko Rosenmüller

geb. am 09.12.1975 in Beckendorf-Neindorf

Gutachter:

Prof. Dr. Gunter Saake,
Prof. Dr. Ulrich Eisenecker,
Prof. Dr. Uwe Aßmann

Ort und Datum des Promotionskolloquiums: Magdeburg, 17.06.2011

Rosenmüller, Marko:
Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines
Dissertation, Otto-von-Guericke-Universität
Magdeburg, Germany, 2011.

Abstract

Scalable software composition by reusing software assets is one of the major challenges in software engineering since many years. The goal of *software product line engineering* is to provide a systematic approach for reuse for a set of similar programs, called a *software product line (SPL)*. The programs of an SPL can be distinguished in terms of *features*, which describe commonalities and differences between the programs. This allows SPL engineers to describe a program by listing its features. There are several SPL development techniques that support composition of assets to derive a tailor-made program according to a selection of required features. A main difference between existing techniques is their support for different *feature binding times*, i.e., the time at which a feature is included in a program. We can distinguish between *static* and *dynamic binding*. While static binding occurs before runtime (e.g., at compile-time), dynamic binding occurs during program start or in a running program.

Both binding times have benefits and drawbacks. Static binding is used to generate tailor-made programs according to the requirements that are known at build-time. By contrast, dynamic binding allows stakeholders to choose required functionality at runtime. This provides high flexibility to tailor a program with respect to available resources and user preferences on demand. For example, for a mobile device we can decide at runtime of a program, which features are actually required according to the location of the device. This reduces resource consumption since only required features have to be loaded. Unfortunately, dynamic binding also has a negative effect on resource consumption due to an overhead to support dynamic changes. Static binding, on the other hand, avoids this overhead and enables optimizations at the source code level (e.g., function inlining). However, this results in limited flexibility because needed features have to be known before runtime. Hence, different binding times are better suited for different application scenarios.

With current approaches for SPL development, a developer is forced to choose between static and dynamic binding at development time. As a result, source code developed for static binding cannot be easily reused for dynamic binding and vice versa. Furthermore, in order to support different binding times for different features of an SPL, programmers must use a mixture of several approaches. For example, dynamically bound components can be statically customized by using a preprocessor-based approach. This combines benefits of both approaches but still hinders reuse of individual features for different binding times. Consequently, *binding time flexibility*, i.e., the ability to flexibly choose the binding time after development is getting increasing attention in research.

In this thesis, we present a combined approach that closely integrates static and

dynamic binding. Our approach is based on *feature-oriented programming (FOP)*, which enables developers to implement the features of an SPL in individual *feature modules*. Based on FOP, we provide means to statically generate tailor-made programs or dynamically compose binary feature modules. However, exclusive use of static or dynamic binding is not sufficient for every application scenario and limits reuse of SPLs. To overcome this limitation, we enable SPL engineers to choose the binding time per feature after SPL implementation. We achieve this with a flexible feature composition mechanism that generates *dynamic binding units* according to the requirements of an application scenario. In contrast to existing solutions that support different binding times, a binding unit integrates multiple dynamically bound features that are always used in combination. At program startup or at runtime, the binding units are composed according to the requirements of the user and the dynamic context. We thus achieve high flexibility by using dynamic binding but also support fine-grained customizations and optimizations by statically generating binding units from a set of features. By using *feature models* for composing binding units, we achieve composition safety at runtime and abstract from the actually used binding units.

Zusammenfassung

Ziel der Entwicklung von Softwareproduktlinien ist die systematische Wiederverwendung von Software innerhalb einer Menge ähnlicher Programme, einer sogenannten *Softwareproduktlinie*. Die Programme einer Softwareproduktlinie unterscheiden sich in ihren *Merkmalen* (engl. *Feature*). Ein Feature beschreibt eine Eigenschaft eines Programms, die relevant für dessen Nutzer oder Entwickler ist. Mit Hilfe von Features können Gemeinsamkeiten und Unterschiede der Programme einer Produktlinie systematisch erfasst werden. Ein Programm kann daher mit Hilfe einer Liste seiner Features beschrieben werden. Es existiert eine Vielzahl von Techniken zur Entwicklung von Softwareproduktlinien. Einige dieser Techniken erlauben es, ein maßgeschneidertes Programm durch Komposition wiederverwendbarer Einheiten entsprechend der benötigten Features zu erstellen. Ein wesentlicher Unterschied zwischen verschiedenen Ansätzen zur Produktlinienentwicklung ist der Zeitpunkt, zu dem die benötigten Features einer Produktlinie in einem konkreten Programm gebunden werden, die sogenannte *Bindungszeit* (engl. *Bindingtime*) eines Features. Bezüglich der Bindingtime kann zwischen *statischem* und *dynamischem* Binding unterschieden werden. Während statisches Binding vor der Programmlaufzeit erfolgt (z. B. zur Übersetzungszeit), findet dynamisches Binding während des Programmstarts oder während der Laufzeit statt.

Für den Nutzer eines Programms ergeben sich je nach verwendeter Bindingtime verschiedene Vor- und Nachteile. Statisches Binding erlaubt es, maßgeschneiderte Programme entsprechend der Anforderungen eines Nutzers oder eines Anwendungsszenarios vor der Laufzeit zu erstellen. Dies bedeutet aber auch, dass die notwendigen Features bereits bei der Erstellung des Programms bekannt sein müssen. Im Gegensatz dazu ist es mit Hilfe des dynamischen Bindings möglich, auch während oder nach dem Programmstart zu entscheiden, welche Features in einem Programm Verwendung finden. Dies erhöht die Flexibilität in Bezug auf die Maßschneiderung, da auch Anforderungen berücksichtigt werden können, die erst zum Programmstart oder zur Laufzeit bekannt sind. So kann etwa auf einem mobilen Gerät entsprechend des Aufenthaltsorts entschieden werden, welche Features zu verwenden sind. Dies wirkt sich zudem positiv auf den Ressourcenbedarf aus, da nur tatsächlich benötigte Features geladen werden müssen. Dennoch wirkt sich dynamisches Binding teilweise nachteilig auf den Speicherbedarf und die Ausführungsgeschwindigkeit von Programmen aus. Dies hat verschiedene Ursachen. Zum Einen ergibt sich ein Overhead für die Realisierung des dynamischen Ladens und Bindings von Features. Zum Anderen verhindert dynamisches Binding statische Optimierungen des gesamten Programms über mehrere Module hinweg. Statisches Binding hingegen vermeidet diesen Overhead und ermöglicht beliebige statische Optimierungen, wie etwa das Funktion-

sinlining durch den Compiler. Aufgrund dieser Unterschiede wird statisches oder dynamisches Binding je nach Anwendungsfall bevorzugt.

Bei Verwendung aktueller Ansätze zur Entwicklung von Produktlinien sind Entwickler gezwungen, sich zwischen statischem und dynamischem Binding zu entscheiden. Das Resultat ist eine eingeschränkte Wiederverwendbarkeit, da Programmcode der für statisches Binding entwickelt wurde, nicht für dynamisches Binding wiederverwendet werden kann. Ebenso kann Programmcode der für dynamisches Binding entwickelt wurde nicht für statisches Binding verwendet werden. Zudem erfordert die gleichzeitige Verwendung beider Bindingtimes die Verwendung unterschiedlicher Technologien zur Softwareentwicklung. So werden beispielsweise Komponentenansätze verwendet, wenn dynamisches Binding notwendig ist und diese mit Präprozessoren kombiniert, um eine statische Maßschneiderung der Komponenten zu erreichen. Dies vereint zwar einige Vorteile beider Ansätze, löst jedoch nicht das Problem der Wiederverwendbarkeit und erhöht die Komplexität des Softwareentwicklungsprozesses. Ziel neuerer Forschungsansätze ist daher die Verwendung flexibler Bindingtimes, um Entwicklern das Festlegen der Bindingtime auch nach der Entwicklung zu ermöglichen.

In dieser Arbeit wird unter Verwendung des Paradigmas der *featureorientierten Programmierung (FOP)* ein Ansatz zur Integration statischen und dynamischen Bindings präsentiert. FOP erlaubt es Entwicklern einer Produktlinie einzelne Features in separaten *Featuremodulen* zu implementieren. Basierend auf einer solchen Implementierung werden im Rahmen dieser Arbeit Ansätze zur statischen Generierung maßgeschneiderter Programme mit Ansätzen zur dynamischen Komposition vereint. Die Integration beider Ansätze erlaubt es Softwareentwicklern nach der Implementierung einer Produktlinie je Feature zu entscheiden, ob statisches oder dynamisches Binding zu verwenden ist. Dies wird durch einen flexiblen Kompositionsprozess erreicht, der es ermöglicht, *dynamische Bindingunits* entsprechend der Anforderungen des jeweiligen Anwendungsszenarios zu generieren. Im Gegensatz zu existierenden Ansätzen integriert eine Bindingunit mehrere dynamisch gebundene Features, die zur selben Zeit verwendet werden. Zum Programmstart oder zur Laufzeit wird eine Bindingunit als Ganzes entsprechend den Nutzeranforderungen und des aktuellen Kontext dynamisch gebunden. Durch die dynamische Komposition von Bindingunits wird die vom Anwendungsszenario benötigte Flexibilität erreicht, wohingegen die statische Generierung der Bindingunits feingranulare Maßschneiderung und statische Optimierungen erlaubt. Durch die Verwendung von *Featuremodellen* während der Komposition wird von verwendeten Bindingunits abstrahiert und eine gültige Komposition zur Laufzeit sichergestellt.

Acknowledgements

Writing a doctoral thesis is a task that requires much time, motivation, and discipline. Sometimes it means much fun and sometimes it means a high burden. It also means further development of ones scientific and even social skills. All this is not possible without cooperation and help of other people.

First of all, I want to thank my family and my friends for support that goes beyond computer science and is often so much more important. I want to especially thank Susanne, for her love, for her continuous support, for reducing my load particularly during the last months, and for motivating me when it was not so much fun. I also want to thank my son Max, who motivated me during the last year without even knowing it. I want to thank my parents, my sister, and my grandmother for their support during all the years of education and for believing in me all the time.

Doing my PhD would not have been possible without the guidance of my advisor Gunter Saake. I want to thank him for the freedom he gave me during that time, for his support when funding was an issue, and for his advice and discussions about my work and many publications.

I started with my PhD one year after finishing my diploma thesis, but actually it had already begun two years before. Thomas Leich and Sven Apel had been my advisors when writing my diploma thesis and later on colleagues at the database working group. Thomas encouraged me to write my diploma thesis about feature-oriented programming, which resulted in FeatureC++. Without this starting point, I would not have been able to continue this work in the past years. After finishing my diploma, Sven was convincing me that it is a good idea to extend my work on FeatureC++ and to start my PhD. Both, Thomas and Sven, then helped me getting started and we had many fruitful discussions about my work.

During the following years at the database working group, my colleagues Martin Kuhlemann and Norbert Siegmund have always been a source for inspiration, which helped me to continue and to finally finish my thesis. I want to thank both for collaboration in many projects and publications and also for many controversial discussions about hot topics of our work. Especially discussions with Norbert were often a seed for new ideas and helped to find solutions for problems.

I also want to thank Christian Kästner, Mario Pukall, Sandro Schulze, and Thomas Thüm, who are, at least partially, also working on FOSD. They had many invaluable comments on my work and contributed to several publications. Even when there was not much overlap in our work, we found topics for cooperation. They had always a helping hand and made the past years a really nice time, including the "fun part". Finally, I want to thank all my other colleagues for numerous helpful comments and discussions about my work and for making the last years a remarkable and highly interesting time.

Contents

List of Figures

List of Tables

Abbreviations

AHEAD	Algebraic hierarchical equations for application design
AOP	Aspect-oriented programming
ATS	AHEAD tool suite
BNF	Backus-Naur Form
CBSE	Component-based software engineering
CORBA	Common Object Request Broker Architecture
DBMS	Database management system
DLL	Dynamically link library
DSPL	Dynamic software product line
EJB	Enterprise Java Beans
FODA	Feature-oriented domain analysis
FOP	Feature-oriented programming
FOSD	Feature-oriented software development
KLOC	Thousand lines of code
LOC	Lines of code
MPL	Multi product line
OOP	Object-oriented programming
SN	Sensor network
SPL	Software product line
SQL	Structured Query Language
UML	Unified Modeling Language

1 Introduction

Building large software systems by composing independently developed modules is one of the major challenges in software engineering [Big98]. Scalable composition of software requires highly *reusable* software assets. Unfortunately, it is hard to achieve a high degree of reuse combined with appropriate effort for composing the assets. Due to an ever increasing complexity of software systems, insufficient reuse leads to inadequate scalability of software development. Solving this scalability issue is one of the challenges of software engineering research since decades [NR69]. *Software product line (SPL)* engineering aims at improving reuse by deriving a set of similar programs from a common code base [CE00]. It tries to overcome limited reuse and scalability issues by providing a systematic approach to reuse and by automating composition of reusable assets. In this thesis, we aim at improving SPL development techniques by increasing the flexibility of the automated composition process. We provide a mechanism for feature composition that supports different binding times. That is, we can compose reusable assets *statically*, before program execution, *dynamically*, at runtime, or using a combination of both mechanisms.

1.1 Overview

Feature-oriented software development (FOSD) subsumes a number of software development techniques that aim at improving reuse by using *features* to model, design, and implement software [AK09]. Features often correspond to user requirements and describe commonalities and variability of the programs of an SPL. Domain engineers describe a concrete program by listing its features. An entire SPL (i.e., the set of valid programs) can be described with a *feature model* [KCH+90]. A feature model structures the features of an SPL and explicitly described dependencies between features to avoid invalid feature combinations.

There are several different software implementation techniques that support feature-oriented development of SPLs. Well known methods include *components* [Sam97], *preprocessors* [KAK08], *collaboration-based* approaches [VN96a], *aspect-oriented programming (AOP)* [KLM+97], *generative programming* [CE00], and *feature-oriented programming (FOP)* [Pre97, BSR04]. In contrast to many other approaches, FOP enables programmers to modularize the features of an SPL into *feature modules* and to derive different programs by composing these modules. This allows software engineers to generate a software system according to the actually required features. In contrast to a component, a feature module implements a single feature. It is composed with other feature modules to derive a concrete program. The point in time at which a feature is *bound* (i.e., included in a program) is called

binding time [CE00]. Depending on the underlying implementation technique, features can be bound at all phases of the software life cycle [ABE+97]. Examples are binding at compile-time or at runtime. We can group the binding times into *static* binding before program execution (as in most FOP approaches) and *dynamic* binding at load-time or in a running program (as in component-based approaches).

The decision whether to use static or dynamic feature binding, influences flexibility and resource consumption of a program. Static binding requires to know which features are used already before deployment, but it enables optimizations at the source code level (e.g., function inlining) [HC02, CRE08]. By contrast, dynamic binding improves a program's flexibility because users can choose required functionality after deployment. This enables composition depending on user requirements at program startup or even according to the dynamic context at runtime. Furthermore, dynamic binding is needed in long-running systems that cannot be stopped but have to adapt to changing requirements [HHPS08b]. However, dynamic binding often increases resource consumption because it introduces an overhead to enable program modifications at load-time or in a running program.

Components and the C preprocessor are frequently used SPL implementation techniques that support dynamic and static feature binding respectively. As a result, both approaches are favored in different domains and application scenarios. For example, the C preprocessor is often used in the embedded domain since it does not introduce any overhead at runtime. Combined with static code optimizations and generative techniques, static binding enables optimizations across feature boundaries. It is thus appropriate for composing many small features, which is important to achieve high customizability. By contrast, components are often too heavy weight for small devices and are only used if resources are not highly restricted. As proposed by Biggerstaff, dynamic binding should only be used at the level of subsystems, i.e., to compose large components [Big98]. This reduces the overhead of dynamic binding and reduces the complexity of composition at runtime at the cost of customizability.

Consequently, programmers seek for approaches to combine different binding times. For example, if a component approach is used for dynamic binding, the C preprocessor can be used for static customization of the components. Unfortunately, this mixture of different approaches forces SPL developers to choose the binding of a feature already before development. Using different binding times for a feature in different program variants is thus not possible. Moreover, source code developed for static binding cannot be reused for dynamic binding and vice versa. To avoid such problems, SPL development techniques should abstract from the used binding time and the binding mechanism [Aßm03]. Hence, *binding time flexibility*, i.e., the ability to choose the binding time after development, is getting increasing attention [ASB+09, CRE08] and is already addressed by a number of scientific approaches [CRE08, EC00, GS05, SE08b].

Besides flexibility, the binding time also influences development and maintenance of SPLs. For example, component approaches force developers to modularize a software system according to the units used for dynamic binding, i.e., the components. That is, the binding mechanism dictates how a system must be decomposed into

modules. This does not necessarily result in the best possible SPL design. For example, small features cannot be modularized because this would degrade performance inappropriately. Hence, a component-based decomposition sometimes contradicts other important concerns of software development, such as implementation effort, software evolution, program comprehension, etc. Moreover, decomposition into components usually conflicts with the features of an SPL, which often cut across many components [Gri00]. To overcome these limitations, we argue that programmers should be able to modularize software independently of the code units used for dynamic binding.

1.2 Contribution

The goal of this thesis is to provide a scalable approach for SPL development that seamlessly integrates static and dynamic feature binding. By using FOP, we allow developers to implement the features of an SPL once and use code transformations to support static binding, dynamic binding, or a combination of both. We thus fill the gap between approaches that support either static or dynamic binding and avoid a mixture of different implementation techniques. We integrate static and dynamic binding by generating *dynamic binding units*. Similar to components a dynamic binding unit contains multiple features and is bound dynamically as a whole. In contrast to components, a binding unit is generated on demand by statically composing a set of features. This allows us to customize and optimize binding units during product derivation. Compared to other approaches that combine static and dynamic binding, we smoothly integrate both binding times: The features of a dynamic binding unit are statically bound with respect to other features of the same binding unit but are dynamically bound with respect to the SPL (i.e., with respect to the features of other binding units). We support the whole spectrum of binding times from static binding of all features to purely dynamic binding by gradually increasing the number of dynamically bound features. Overall, we provide means to improve several aspects of SPL development:

Implementation: We simplify SPL development by providing a single implementation mechanism for features that abstracts from binding details.

Flexibility: We enable programmers to flexibly decide per application scenario which binding time to use. With static binding, we achieve fine-grained customizability; with dynamic binding we provide flexibility at runtime.

Resources: We optimize resource consumption (e.g., CPU utilization and memory consumption) by enabling programmers to define binding units per application scenario and by dynamically composing binding units according to requirements at runtime.

Reuse: We improve reuse of features by enabling a fine-grained decomposition and by supporting different binding times for the same feature.

Safety: We support safe composition of features statically and dynamically by using feature models for validating configurations. We reduce the probability of runtime errors by using static binding within dynamic binding units.

Scalability: We achieve scalability of static and dynamic binding by applying our approach to generate tailor-made components that can be used to build larger software systems.

In the following, we provide a detailed overview of our contributions.

Static Feature Binding. Resource consumption and high customizability are especially important for the domain of embedded systems, in which resources such as working memory, power supply, and computing power are limited. Applying FOP to such domains means that we have to support fine-grained customization and that we must not introduce any overhead with respect to resource consumption. However, there is less known about customizability and resource consumption of FOP, e.g., compared to a preprocessor-based implementation of an SPL. We thus provide an optimized composition mechanism for FOP that does not degrade performance or memory consumption. In a case study, we evaluate the approach and compare it to an SPL implementation with the C preprocessor. We show that decomposing software into feature modules does not introduce any overhead with respect to performance or application footprint. This allows us to apply FOP to a wide range of application domains from embedded devices to large scale systems.

Dynamic Feature Binding. Based on FOP and static feature binding, we present an extended approach that allows us to bind the features of an SPL dynamically. In contrast to a mixture of different implementation techniques, we support static and dynamic binding for the same implementation of a feature, i.e., for the same code base of an SPL. Because the SPL configuration process is a complex task, a valid configuration of statically bound features is usually enforced by tools [Bat05, pur04, Kru08]. These tools consider configuration constraints, feature interactions, and crosscutting concerns when deriving a concrete configuration. When dynamically composing features, these feature dependencies have to be considered as well. Hence, we have to derive a configuration at runtime that corresponds to the SPL's feature model. Because external tools are often inappropriate for this task, we provide a generative approach for creating an SPL-specific infrastructure for instantiation and validation of products at runtime.

Dynamic Binding Units. Existing approaches that support binding time flexibility allow a feature to be bound either statically or dynamically. We go one step further and compose a set of dynamically bound features statically into a single *dynamic binding unit*. We generate a dynamic binding unit according to the requirements of a user or an application scenario: Features that are always used in combination are merged into a binding unit before compilation and can be statically optimized. In

a running program, the generated binding units are composed according to the requirements at runtime. Our approach provides the needed flexibility due to dynamic binding and achieves high customizability and resource efficiency by statically generating binding units from smaller features. We integrate static and dynamic binding also at a conceptual level by transforming the SPL's feature model according to generated binding units. The resulting model contains only dynamic variability and we use it to achieve safe composition of dynamic binding units. We evaluate the combined approach regarding resource consumption and flexibility to demonstrate the benefits of dynamic binding units. We show that there is room for optimizations by allowing programmers to choose the optimal trade-off between static and dynamic binding per application scenario.

Runtime Adaptation with Binding Units. Finally, we demonstrate the practical relevance of our approach by the example of *dynamic SPLs (DSPLs)* [HHPS08a]. In contrast to a traditional SPL, a DSPL adapts to dynamically changing requirements by reconfiguring itself. We show that dynamic binding units allow us to integrate both approaches by generating a DSPL from the features of an SPL. That is, we are able to (1) statically select the functionality required for a DSPL, (2) choose which features are bound statically and which dynamically, (3) define binding units for dynamically bound features, and (4) compose and *reconfigure* a concrete program at runtime. For autonomous reconfiguration, we present a feature-based approach for (self-)configuration. Our approach not only smoothly integrates traditional and dynamic SPLs but also improves runtime adaptation in DSPLs by reducing the compositional complexity.

Scalability of Static and Dynamic Feature Binding. To achieve scalability of the static and dynamic approaches, we analyze the applicability of FOP to develop *component SPLs*. A component SPL is used to generate tailor-made components that are composed to derive larger software systems. Using FOP for component development, results in components with a variable interface. When using static binding, clients have to handle reconfiguration of components and have to be able to use multiple differently configured components at the same time. When using dynamic binding, clients have to be able to use components that are reconfigured during runtime. We thus introduce the notion of *semantic* and *programming interfaces* of SPLs and suggest how to handle component variability in clients when using static and dynamic binding.

1.3 Outline

We evaluate the concepts presented in this thesis with FeatureC++[1], an extension of the C++ programming language that supports FOP [ALRS05]. The following chapters are structured as follows:

[1] http://fosd.de/fcc

Chapter 2. In the next chapter, we provide background on SPL engineering and feature binding. We focus only on domain and application engineering processes that are addressed by the concepts we introduce in this thesis. Furthermore, we provide an overview of binding times important for SPL development.

Chapter 3. We analyze the requirements for binding time flexibility in Chapter 3. We review existing approaches for SPL development with respect to supported binding times and customizability. Finally, we derive detailed goals for this thesis.

Chapter 4. In Chapter 4, we describe optimizations for the static code generation process in FeatureC++ to avoid any overhead for feature composition. By means of a case study, we evaluate static binding in FeatureC++ and compare it to the C preprocessor. We introduce SPL interfaces and propose solutions to handle variability of statically generated components.

Chapter 5. For dynamic binding of feature modules, we present code transformations for FeatureC++ in Chapter 5. We support dynamic binding of the features of an SPL for the same code base that is used for static binding. Furthermore, we provide a mechanism for validating configurations in a running program by using feature models. Finally, we describe how client programs can handle dynamic variability of component SPLs.

Chapter 6. We present an approach that integrates static and dynamic binding in Chapter 6. We introduce a flexible composition mechanism that allows us to generate tailor-made dynamic binding units. To preserve safe composition using feature models, we transform an SPL's feature model according to the statically generated binding units. We evaluate the approach with two case studies.

Chapter 7. In Chapter 7, we provide an extension of the presented approach that allows us to generate tailor-made DSPLs. With an approach for feature-based (self-)configuration we propose a mechanism for reconfiguration of DSPLs that is independent of the used binding units. In a case study, we demonstrate that dynamic binding units can improve runtime adaptation in DSPLs.

Chapter 8. In the last chapter, we conclude and describe challenges to further improve feature binding in SPLs and to closely integrate the presented work with component-based software development.

2 Software Product Lines and Feature Binding Time

In this chapter, we provide background information with respect to software product lines. We describe mechanisms and terminology of *software product line engineering* and *feature binding* that are important for the concepts we introduce in this thesis.

2.1 Software Product Line Engineering

According to *product lines* in other industries, a set of similar software products is called a *software product line (SPL)*. Consequently, also the software industry aims at a reduction of production costs by reusing *software assets* when building the products of an SPL. This seems to be natural because the products of an SPL share commonalities that offer reuse opportunities. However, reuse is not achieved automatically; it requires a reuse-centric software development process, which is the aim of *software product line engineering* [CE00].

2.1.1 Software Product Lines

Withey defines a product line as *"a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market"* [Wit96]. The programs of an SPL are thus distinguished in terms of *features*, which are (functional or non-functional) properties that are relevant to some stakeholder [CE00][1]. For example, a product line of database management systems (DBMS) may consist of a full-fledged DBMS for online transaction processing, a medium size DBMS for web databases, and a small DBMS for mobile devices. We can distinguish the different systems by listing their features, such as a feature for transaction management and a query processor. Large features of an SPL can be described in terms of smaller features such as the lock protocols used to implement a transaction management system. This allows us to provide an arbitrary detailed description of the functionality and the non-functional properties of an SPL.

In contrast to an SPL, a *program family* [Dij72b] (or more generally a *product family*) is a set of programs that is built from a common code base. Withey defines a product family as *"a group of products that can be built from a common set of assets"* [Wit96]. This definition does not apply to product lines whose products are developed independently. However, it is possible to develop an SPL as a program

[1] See Section 2.1.4 for a definition of features as we use it throughout this thesis.

family [CE00], which provides high reuse opportunities. For example, we can develop the products of a DBMS SPL independently or by composing a set of software components [DG01a]. In literature, the terms product line and product family are often used interchangeably. In this thesis, we use the term software product line (SPL) and assume that an SPL is developed as a family of programs. When it is important to distinguish between SPL and program family we will make this explicit.

In contrast to the development process for single software products, the SPL development process is highly focused on reuse of software assets to reduce development costs. Parnas already recognized the importance of analyzing the commonalities of a whole program family before focusing on individual family members [Par76]. Similarly, the SPL development process is usually split into *domain engineering* and *application engineering* [CE00]. The aim of domain engineering is to analyze and develop reusable assets for a particular application domain. These assets are used to build concrete software products in the application engineering phase. In the following, we introduce the basic concepts of domain and application engineering. For a more detailed description we refer to [CE00] and [PBvdL05].

2.1.2 Domain Engineering

The domain engineering process can be split into *analysis*, *design*, and *implementation* [CE00].

Domain Analysis. The goal of domain analysis is to find commonalities and differences of possible products of an application domain. While commonalities among different products are the basis for reuse, the differences provide means to distinguish the products from each other. In *feature-oriented domain analysis (FODA)* [KCH+90], a special form of domain analysis, commonalities and variability of an application domain are described in terms of *features*. *Feature modeling* refers to the process of structuring an application domain by describing relations between the features of a domain.

Domain Design. After defining commonalities and variability of the application domain, a domain engineer defines the architecture of an SPL [CE00]. In contrast to the usual software design process, the variability of the product line has to be considered because not a single product but a set of products has to be derived from a common code base. Hence, the domain engineer has to define how variability can be realized (e.g., using components). This also means that the architecture itself may be variable and that different products of an SPL may have a different architecture.

Domain Implementation. In the domain implementation process, an SPL engineer implements reusable assets according to the developed SPL design. There are several implementation techniques that can be used to implement the variability required for an SPL. Examples are the C preprocessor, components, or *feature-oriented programming*, which is in the focus of this thesis.

At all stages of domain engineering, different kinds of software artifacts are developed. This includes documentation, components, test cases, etc. In this thesis, we address all phases of domain engineering but focus on domain implementation, i.e., the development of reusable software assets. We analyze different implementation techniques, we show how feature-oriented programming can be used to implement an SPL, and we provide flexible mechanisms for product derivation, which is part of application engineering.

2.1.3 Application Engineering

The application engineering process is based on the software artifacts developed during domain engineering. It starts with *requirements analysis* which maps customer requirements to the features of an SPL, e.g., defined in a feature model [CE00]. The resulting set of features defines the characteristics of the product to derive. In the *product configuration* phase, the features are mapped to reusable assets developed during domain engineering. Finally, an SPL *instance* (i.e., a concrete product) is *generated* or *composed* from the selected assets. A product has to be customized during application engineering if required features are not part of the SPL or if there are no reusable assets that implement the required features. For example, new assets may be developed for a customer of a DBMS who requires a special index structure for fast data access. The new features and the corresponding assets may be integrated in the code base of the SPL to reuse them in future products.

In the following, we focus on the product configuration and code generation phases of the application engineering process. We present different code transformation mechanisms that are based on the same code base but provide flexibility during product derivation.

2.1.4 Feature-oriented Software Development

In this thesis, we focus on *feature-oriented software development (FOSD)* [AK09] to develop SPLs. FOSD considers features as units for modeling and composition of software. For SPL development this means that features are considered in all phases of domain and application engineering. For example, FODA aims at analyzing the features of a domain and *feature-oriented programming (FOP)* aims at implementing these features in distinct *feature modules* [Pre97, BSR04]. In application engineering, features are used for configuration and feature modules are composed when deriving a concrete product from feature-oriented source code.

Features and Variation Points

Features and variation points are central parts of FODA but often defined differently. In the following, we define both as we understand the terms in the context of this thesis.

Features. The term *feature* is used in many different ways in SPL engineering. In this thesis, we use the definition of Czarnecki et al. [CE00]:

> *A feature is a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept.*

This definition is based on the meaning of the term feature in *Organization Domain Modeling (ODM)* [SCK+96]. It goes beyond the definition of a feature by Kang et al., who limit the scope of a feature to a *user-visible characteristic of a system* [KCH+90]. According to the definition of Czarnecki et al., a feature is not necessarily user-visible. It may thus include *internal variability* of a system such as design decisions [PBvdL05].

The definition of a feature as a *characteristic of a concept* means that a feature may describe functionality of a system such as the transaction management of a DBMS, but it may also describe a *non-functional property* such as performance or quality of service. In the context of FOP, a feature usually describes a unit of functionality [AK09]. The distinction between functional features and non-functional properties is important since we cannot implement every non-functional property (and thus not every feature) in a modular unit. The reason is that some non-functional properties only *emerge* due to the interplay of several software assets in a concrete program and sometimes only during runtime [SRK+08]. They are also called *emergent properties* of a system and are mostly not implemented as separate modules in FOP.

Variation Points. In FODA, domain engineers analyze commonalities and variability of the products of an SPL. While commonalities are included in many or all products, variable parts are only included in selected products. Hence, a product is usually distinguished from other products in terms of the variable parts. For this reason, *variation points* are used to document the variability of an SPL in variability models [PBvdL05]. Pohl et al. define a variation point as follows [PBvdL05]:

> *A variation point is a representation of a variability subject within domain artefacts by contextual information.*

Variation point and feature thus mean different things. Griss et al. distinguish between mandatory features, optional features, and variant features [GFdA98]. Variant features represent variation points; they describe alternative ways to configure mandatory and optional features. Mandatory features are the same for a number of products and optional features are not required for every product.

Feature binding. *Feature binding* defines when and how variant and optional features are selected during the SPL life cycle [SCK+96]. The time at which a feature is bound is called *feature binding time* [KCH+90]. We analyze the binding time in detail in Section 2.3. Selecting the features of an SPL that are used in a concrete program is also called *product configuration* and may be done manually or automatically [CE00]. A concrete product can thus be described by a list of features.

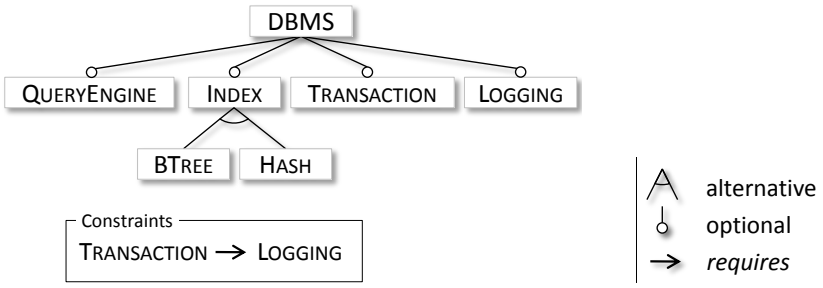


Figure 2.1: Feature model of a DBMS.

Feature Modeling

A *feature model* describes variability and commonalities of an SPL using features and constraints between them [CE00]. A feature model can be visualized in a feature diagram, which is a hierarchical representation of its features. An example of a feature diagram for a DBMS is shown in Figure 2.1. Mandatory and optional features are denoted by filled and empty dots. A mandatory feature must be included in a configuration whenever its parent feature is included. Relations between features and *cross-tree constraints* limit the variability of a feature model to ensure that only valid configuration can be derived. An *or*-constraint defines that at least one of a set of features must be selected, an *alternative* relation between a set of features means that exactly one of them must be selected. *Requires* (a feature requires another feature) and *mutual-exclusion* (two features cannot be used in combination) constraints often suffice to describe cross-tree dependencies. However, arbitrary propositional formulas may be used as constraints [Bat05]. Since the features of an SPL can be bound at different points in time, a feature's binding time is sometimes annotated in the feature model [vGBS01].

Feature Interactions

The features of a program are often not independent, which may result in *feature interactions* [BDC+89]. That is, two features that properly work when used independently may exhibit unexpected behavior when used in combination. If possible, interactions should already be recognized and documented at the modeling level to be able to handle them at implementation level. There are several ways to handle feature interactions in SPL development [LBL06a, AK09] and we provide a more detailed overview about implementation of feature interactions in the next section.

2.2 Product Line Implementation

The reusable assets of an SPL are developed in the *domain implementation* phase of domain engineering. Well known concepts for implementing SPLs are annotation-based approaches such as preprocessors [Käs10], components [KCH+90, HC01],

collaboration-based designs [VN96a], C++ templates [CE00], *aspect-oriented programming (AOP)* [KLM+97], and *feature-oriented programming (FOP)* [Pre97, BSR04]. In the following, we describe selected implementation techniques important for this thesis.

2.2.1 Separation of Concerns

Separation of concerns is a principle important for software development and especially for implementing SPLs. It means to separate different concerns from each other to be able to focus on selected concerns only, e.g., to be not disturbed by a particular concern [Dij82]. In software development, a concern can be any important aspect such as a feature of an SPL or a class in OOP. Dijkstra already recognized that separation of concerns is not always possible for all kinds of concerns at the same time. This is also known as the *tyranny of the dominant decomposition* [TOHSMS99]. It means that one usually has to decide which concern to separate, such as a class in OOP, because different concerns are not independent. Parnas furthermore noticed that when decomposing a system into modules (a.k.a. *modularization*) there is usually a criterion that results in a more efficient decomposition than other criteria [Par72]. *Multi-dimensional separation of concerns* (MDSoC) tries to separate multiple concerns at the same time by describing the concerns as different views on the system [TOHSMS99].

A feature of an SPL can also be seen as a concern that can be separated from other features (i.e., other concerns), which is the aim of FOSD. This is sometimes hard to achieve because individual features often crosscut the entire implementation of a program. For example, a feature usually crosscuts multiple classes in OOP. In general, a concern that cuts across other concerns is called a *crosscutting concern* [KLM+97]. This also applies to a feature of a program that crosscuts other concerns such as several classes or even other features. For example, a feature TransactionManagement of a DBMS often crosscuts large parts of the source code of a DBMS because it cannot be aligned with other concerns of the software such as the classes used to implement it.

2.2.2 Annotative and Compositional Approaches

Source code annotations are used to implement SPLs by annotating the code that implements a particular functionality of an SPL. It is called *annotative approach* for implementing variability [Käs10]. Examples for annotative approaches are the C preprocessor [LAL+10] and the CIDE approach of using colors to annotate code fragments [Käs10]. In contrast to annotative approaches, *component-based software engineering (CBSE)* [KCH+90, HC01] and FOP aim at *modularizing* functionality in reusable entities that are *composed* to yield a tailor-made program. They are called *compositional approaches* for SPL development.

Modularization is understood as an important factor for improving program comprehension [Dij76, Par79], reuse [Big98], and other aspects of software development.

In SPL development, modularization of functionality is important for composing different programs from a common code base. First, when we are able to implement an SPL in separate modules, product derivation means composition of these modules. Furthermore, when a module implements exactly one feature, composition of the modules corresponds to composition of features (which is the aim of FOP) and we have direct control of the features in a program.

With respect to modularization, the C preprocessor and components mark two ends of the spectrum of implementation mechanisms for SPLs. While annotative approaches do not aim at modularization of functionality, components are stand-alone units of reuse. CBSE thus allows for developing SPLs by separating functionality of a system in reusable modular components [HC01]. A component usually encapsulates multiple features [LK06]. By contrast, annotative approaches allow for implementing an SPL by annotating the features of an SPL. In the following, we describe the general concepts of annotational and compositional approaches with the example of the C preprocessor and CBSE. We finally describe how FOP is used for SPL development.

2.2.3 The C Preprocessor

Berkeley DB[2] and SQLite[3] are two examples for DBMS that provide customizability of application functionality using the C preprocessor. A preprocessor operates on the source code of a program in a *preprocessing* step of the build process. The C preprocessor is a text-based preprocessor and enables transformations of a program such as removing annotated parts of the code or replacing code fragments with more specialized variants.

The C preprocessor adds preprocessing functionality to the C programming language [KR88]. Since it is integral part of every C/C++ compiler it can be easily used for software development with C or C++. It is commonly used to customize a program with respect to the underlying operating system, compiler, and hardware, i.e., to different execution environments [Fav96]. Such programs can also be seen as SPLs that implement *external variability* of the environment. Customizations with respect to external variability are even important for platform independent languages. For example, development of Java applications for embedded systems is not possible without considering restricted resources and very heterogeneous hardware. This often means that code has to be customized for different platforms (e.g., using the Antenna preprocessor[4]).

Preprocessors are especially used for SPL development in the embedded domain because they provide a simple mechanism for implementing variability without any overhead at runtime. That is, annotations are used to mark code that implements functionality important for a specific variant of the SPL [Käs10, LAL+10]. Especially for the C preprocessor this is an error-prone process. Syntactic correctness and type

[2] http://www.oracle.com/technetwork/database/berkeleydb/
[3] http://www.sqlite.org/
[4] http://antenna.sourceforge.net/wtkpreprocess.php

```
1  #ifdef DEBUG
2  #define DEBUG_LOG(args) printf(args)
3  #define PRECOND(exp) if (!exp) {\
4    DebugBreak(); return false; };
5  #else
6  #define DEBUG_LOG(args)
7  #define PRECOND(exp) if (!exp) return false;
8  #endif
```

```
9  class DB {
10   bool Put(Key* key, Value* val) {
11     PRECOND(key!=NULL);
12     DEBUG_LOG("DB.Put",key);
13
14 #ifdef TRANSACTION  //feature Transaction
15     Txn* txn = BeginTransaction();
16 #endif
17
18     ... //data storage of key value pairs
19   }
20
21 #ifdef TRANSACTION  //feature Transaction
22   Txn* BeginTransaction() { ... }
23 #endif
24 };
```

Figure 2.2: C++ source code of class `DB` of a DBMS with preprocessor-based annotation of feature Transaction.

safety can be enforced with a special preprocessor that supports only disciplined annotations and provides a type system [Käs10].

Preprocessors thus provide means for a feature-oriented implementation of an SPL by annotating the features in the source code. During product configuration, the feature selection is mapped to a set of annotated code fragments that remain in the code. Code fragments that are annotated but have not been selected are not included in the generated program (a.k.a. *conditional compilation*). An excerpt of a DBMS product line implemented with the C preprocessor is depicted in Figure 2.2. Shown is a class `DB` that is annotated with `#ifdef` preprocessor statements. Code surrounded by annotations is only included in a concrete program when the according variable (e.g., `TRANSACTION`) has been defined. We can thus derive variants of our DBMS with or without the Transaction feature. Such annotations can be used to annotate arbitrary code fragments such as single statements (Lines 14–16), whole methods (Lines 21–23), or classes.

Another mechanism used for SPL development are macros, which are *expanded* by the C preprocessor with code defined at a different place. For example, macro `DEBUG_LOG` in Line 12 of Figure 2.2 refers to a macro that is expanded according to its definition in a different file (Lines 2 and 6). By combining macros and with conditional compilation, a programmer can provide different definitions of a macro that depend on conditional preprocessor statements. Actually, this is the way pre-

```
1  bool Precond(bool condition) {
2  #ifdef DEBUG
3      if (!condition)
4        DebugBreak();
5  #endif
6      return condition;
7  };
```

```
8  class DB {
9    bool Put(Key* key, Value* val) {
10     if (!Precond(key!=NULL))
11       return false;
12     ... //actual data storage
13   }
14 };
```

Figure 2.3: C++ source code of class `DB` when replacing macros with method calls.

processor statements should be used in practice to increase readability of the source code [SC92]. In our example, the macro expands to a method for logging information in a debug variant of the program. In a release variant it expands to an empty statement, which removes the code from the program completely. In contrast to a method, C preprocessor macros do not provide their own scope of execution and can introduce statements that modify existing variables or change the control flow.[5] For example, macro `PRECOND` in Line 11 aborts execution of the method by introducing a return statement when the given condition is not satisfied (Lines 3 and 7). When replacing the macro with a method (as shown in Figure 2.3), we have to introduce the conditional and the return statement around the method call (Line 10). That is, we have to repeat the same code (`if (...) return ...`) at every place we want to achieve this behavior, which is error-prone and degrades maintainability of the code.

Code Reuse. Annotations are especially important for SPL development because they can be used to remove source code that is scattered across the whole program depending on a configuration option. This enables a programmer to write source code that can be configured with respect to crosscutting concerns by enclosing all code of that concern with the same annotation. One of the drawbacks of this solution is that the annotated code is scattered all over the program. This negatively affects software development in general [Fav96] and especially SPL development [Käs10]. We will analyze this in more detail in Chapter 3. At this point, note that annotations decrease reuse possibilities because the annotated code can only be used in the context of the code it is embedded in.

Macros, on the other hand, provide better reuse capabilities. They can be defined

[5] A macro may unintentionally modify existing variables, which can be avoided with *hygienic macros* [KFFD86]. Here we do not consider this issue and only focus on modifications of the control flow that can be useful in several situations.

in separate files and reused in any place in the source code. The reuse mechanism is similar to a function call but it does not result in additional code for the call itself because the expanded macro is inserted in-place. This does not introduce an overhead for macro execution but may increase the code size because the macro code is copied to all places where it is used. Macro expansion may also reduce the code size when the macro is empty or when the expanded code is used only once. Hence, macro expansion is similar to method inlining accomplished by compilers. We provide a comparison of macros to other mechanisms for SPL development in Chapter 3. For a more detailed comparison of annotation based approaches and compositional approaches we refer to the work of Kästner [Käs10].

2.2.4 Component-Based Software Engineering

In contrast to a set of source code fragments annotated with the C preprocessor, a component is a self-contained software artifact [Sam97]. Szyperski defines a component as follows [Szy02]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

According to this definition, a component can be deployed independently, which is a main difference to a feature implemented with the C preprocessor. Hence, components are physically separated code units and thus comply with the principle of separation of concerns. The components are connected via *interfaces* [PBvdL05]. A component defines a *required interface* and a *provided interface* to describe which functionality the component uses and which functionality the component offers to other components. There are a number of different kinds of components that comply with the component definition above. It ranges from very large to very small components. For example, a DBMS can be seen as a component that is executed as a separate process and accessed via SQL. The DBMS is composed with other large components, e.g., to build an even larger banking software system. But a component may also be a single class that provides an interface and can be composed with other classes, a.k.a. *code-level components* [MDK96]. In this thesis, we focus on components that are implemented as a single class or a cluster of classes.

The component concept is often associated with a *component model* such as *Common Object Request Broker Architecture (CORBA)*, *Component Object Model (COM)*, *Enterprise Java Beans (EJB)*, which manages composition of components [Sam97]. In the following, we do not limit the term component to components that are used with an object model, which is only a special case. When we do not mention a particular composition mechanism, any mechanism including existing component models may be used.

Class DB

```
1  class DB {
2    virtual bool Put(Key* key, Value* val) {...}
3  };
```

Class DB_TXN

```
4  //feature Transaction
5  class DB_TXN : public DB {
6    Txn* BeginTransaction() { ... }
7
8    virtual bool Put(Key* key, Value* val) {
9      //transaction specific code
10     Txn* txn = BeginTransaction();
11     bool res = DB::Put(key,val);
12
13     if (res)
14       txn->Commit();
15     else
16       txn->Abort();
17     return res;
18   };
19 };
```

Figure 2.4: C++ source code of extension of a class DB of a DBMS via inheritance and virtual methods.

Component Implementation with OOP. There are several ways to implement a component. In the following, we describe how components can be implemented with OOP. As an example, we use a DBMS component and a component for transaction processing, i.e., a component that implements feature TRANSACTION of Figure 2.1. Implementing a single feature with a component is only one way to develop an SPL with a component-based approach. There are more possibilities for mapping features to components [KCH+90, TRCP07].

Our example DBMS provides core functionality such as a method `Put` for storing data as shown in Line 2 of Figure 2.4. We omit any interface description and only provide the implementation of a class `DB`, which implements the DBMS core. In Lines 4–19, we depict the transaction component, implemented by extending class `DB` of the DBMS component.

In our example, the transaction component encapsulates data storage with a transaction. It extends method `Put` (Line 8) and adds transaction specific code before and after the original method. A client can call the virtual method of class *DB* and the correct implementation will be chosen according to the actual type of the object (i.e., method `DB::Put` or method `DB_TXN::Put`).

The implementation of the transaction component in Figure 2.4 is achieved via subclassing `DB` and overriding its virtual method `Put`. Whenever a client needs a DBMS with transaction management it thus has to use an instance of class `DB_TXN` instead of an instance of class `DB`. However, this means that the client has to create the corresponding instance when needed. This can be achieved with a component approach by using the *abstract factory* design pattern to instantiate every class that

may be extended by another component [GHJV95]. An extending component can override the factory methods to create the correct class instance. However, it can be laborious and error-prone to achieve consistency of class instantiations.

Subclassing has to be avoided when the extended classes are unknown (i.e., when only a set of class interfaces are provided). This can be achieved with the *decorator* design pattern [GHJV95], which uses delegation to connect independently developed class fragments. In combination with factory methods component implementation can get very complex.

In our example, the extension mechanism is based on late binding of virtual methods. Depending on the programming language and runtime environment, the correct method implementation will be determined at method execution time (e.g., in Java) or during class instantiation (e.g., in C++ stored in virtual function tables) [Lip96]. However, it is also possible to use static binding for component composition (e.g., binding at link time). We present details about the binding time in Section 2.3.

Hook Methods. When using OOP to implement a component, the points at which a component can extend an existing class are the public and protected methods of that class. Using only public and protected methods as extension points reduces the complexity of a component's interface but is sometimes too restrictive. For example, it is not possible to introduce statements into an existing method as it is possible with the C preprocessor. To provide a more fine-grained extension mechanism, programmers use empty *hook methods* to define additional extension points, e.g., using the *template method* design pattern [GHJV95, HK00]. In Figure 2.5, we depict an example of a hook method `BeginPut` defined in class `DB`. By calling a hook, the programmer of a class allows others to introduce code into the methods that call the hook (Line 6 in Fig. 2.5); by overriding the hook method in a subclass (Line 13), a component defines additional code that is executed when the hook is called.

Implementing Crosscutting Concerns. A component usually implements one or more features [KCH+90, LK06] and a feature may be scattered across many components [KLM+97]. Consequently, there is an m-to-n mapping of features to components. Nevertheless, it is often possible to implement a single feature with a component, as in our DBMS example. This is trivial when the separation of other features is not required, but it can get complicated when we have to modularize crosscutting features.

When two features cut across each other, it is hard to implement both in distinct components that can be deployed independently. The reason is that features may interact and special interaction code is needed to ensure proper behavior of both features in combination. This additional *glue code* or *derivative code* [LBN05] must be executed only when both features are deployed. A solution is to include the interaction code in one of the components and execute the code only when both components (i.e., both features) are active. This, however, does not comply with the principle of separation of concerns. Another solution is to implement the derivative

Class DB

```
1  class DB {
2    //empty hook method
3    virtual void BeginPut(Key* key, Value* val) { }
4
5    virtual bool Put(Key* key, Value* val) {
6      BeginBut(key,val);  //call hook
7      ...                 //actual method impl.
8    }
9  };
```

Class DB_TXN

```
12 class DB_TXN : public DB {
13   virtual void BeginPut(Key* key, Value* val) {
14     ... //transaction specific code
15   };
16 };
```

Figure 2.5: C++ source code of class DB that defines a hook method and class DB_TXN that overrides the hook.

code in a separate component. Either solution causes a high implementation effort to realize component composition. Furthermore, the solutions are error prone since the programmer has to ensure that the code is executed whenever the corresponding components are used. In Section 2.2.5, we show how composition of interaction code is automated in FOP and in Chapter 6, we demonstrate how we can automatically apply interaction code also when using dynamic binding.

2.2.5 Feature-oriented Programming

Feature-oriented programming (FOP) is a programming paradigm that aims at modularizing the features of a program [BSR04, Pre97]. As component-based development, FOP is also a compositional approach. In contrast to other SPL implementation mechanisms, its goal is to generate programs by selecting the required features. That is, there is ideally a one-to-one mapping of features in a domain model and corresponding *feature modules* (i.e., the implementation units). This not only simplifies the product derivation process but also improves the traceability of a feature to its implementation, maintenance, etc. [AMGS05, AK09].

Feature Modules

A *feature module* implements a feature as an increment in program functionality [BSR04]. That is, a feature module extends a base program with additional code similar to what Parnas proposed for program family development [Par76]. In FOP, feature modules are developed and maintained separate from each other to comply with the principle of *separation of concerns* [Dij82]. When composing a program, selected feature modules are usually applied in a step-wise manner using a program generator. The generator composes a feature module with a base program

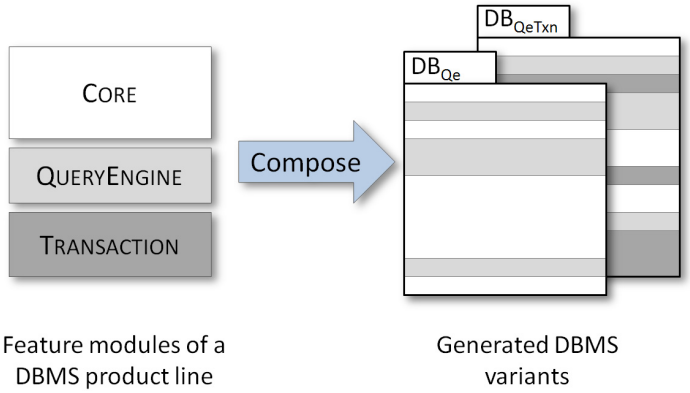


Figure 2.6: Generating two different DBMS by composing feature modules with FOP.

which results in a modified program that is the input for the next composition step.

FOP also supports implementing alternative features to generate programs specialized for an application scenario and a user's requirements. To generate a tailor-made program a user selects a subset of the available features. In contrast to inheritance-based extension of OOP, feature modules can be freely composed (according to domain and implementation dependencies) to derive different programs, which is the reason for the scalability of the approach. While inheritance requires defining the class that is extended already at development time, a feature module may be composed with other modules that are not known during development (as also supported by mixin-based inheritance).

A simple example of a feature-oriented DBMS design is given in the left part of Figure 2.6. It shows a base program and two features QUERYENGINE and TRANSACTION. Basic database functionality is implemented in module CORE and is extended by modules QUERYENGINE and TRANSACTION that implement a query processor and a transaction management system of a DBMS. Based on the modular implementation of features, different variants of concrete DBMS (right part of Figure 2.6) can be generated automatically by composing the required feature modules. In our example, we can generate a simple DBMS only consisting of the CORE implementation, variants that include either QUERYENGINE (DB_{Qe}) or TRANSACTION and a variant that includes all features (DB_{QeTxn}).

Decomposition of Classes

In object-oriented program code, classes and methods are the primary concerns for decomposing a system. However, a feature usually crosscuts several classes. Consequently, the classes have to be decomposed with respect to the features of the software when applying FOP on top of OOP. In Figure 2.7, we depict a decomposition of three DBMS classes `DB`, `Txn`, and `QueryProcessor` with regard to the

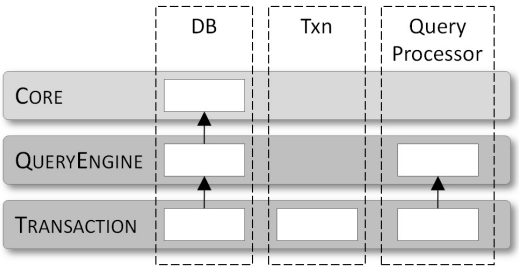


Figure 2.7: Decomposition of classes (dashed rectangles) along features (horizontal layers). Resulting base classes and refinements shown as white rectangles.

features of Figure 2.6. That is, for every class, basic functionality is separated from feature-specific functionality. Classes in module BASE are *refined* in features QUERYENGINE and TRANSACTION, which we denote with an arrow. In this example, the basic implementation of class `QueryProcessor` is refined to implement feature TRANSACTION. By contrast, class `DB` is refined in two features. Feature TRANSACTION cuts across the entire source code of the DBMS.

Implementing Feature Modules

There are several approaches to implement an SPL with respect to features. As a precursor of FOP, the language P++ (an extension of C++) [BDG+94] is based on the GenVoca idea of composing *layers* to build large software systems [BO92]. Composition of roles with C++ templates and *mixin-based inheritance* [BC90] was proposed by VanHilst and Notkin [VN96b] and builds on *collaboration-based* or *role-based* designs [RAB+92]. Mixin-based inheritance is similar to multiple inheritance but avoids some of its problems by linearizing the inherited base classes. *Mixin layers* combine large scale composition of GenVoca layers with mixin-based inheritance for the classes within a layer [SB98, SB02]. This enables composition of entire collaborations (i.e., *layers*) including the contained roles. A mixin layer is implemented as an OO class that extends another layer using mixin-based inheritance. The inner classes of a mixin layer correspond to FOP's class refinements or the roles in collaboration-based designs. An inner class of a mixin layer is composed with an inner class of the base layer using inheritance of nested classes. In current FOP approaches (e.g., in AHEAD [BSR04]), the implementation of a feature has been simplified by removing the enclosing classes that represent features. One reason is that a feature not only contains program code but also several other software artifacts such as documentation, test cases, etc. This concept is implemented in the AHEAD tool suite (ATS)[6], FeatureC++[7], and FeatureHouse[8]. These implementa-

[6] http://www.cs.utexas.edu/users/schwartz/ATS.html
[7] http://fosd.de/fcc
[8] http://fosd.de/featurehouse

Feature CORE

```
1  class DB {
2    bool Put(Key& key, Value& val) { ... }
3  };
```

Feature QUERYENGINE

```
4  refines class DB {
5    QueryProcessor queryProc;
6    bool ProcessQuery(String& query) {
7      return queryProc.Execute(String& query);
8    }
9  };
```

Feature TRANSACTION

```
10 refines class DB {
11   Txn* BeginTransaction() { ... }
12
13   bool Put(Key& key, Value& val) {
14     ... //transaction specific code
15     return super::Put(key,val);
16   };
17 };
```

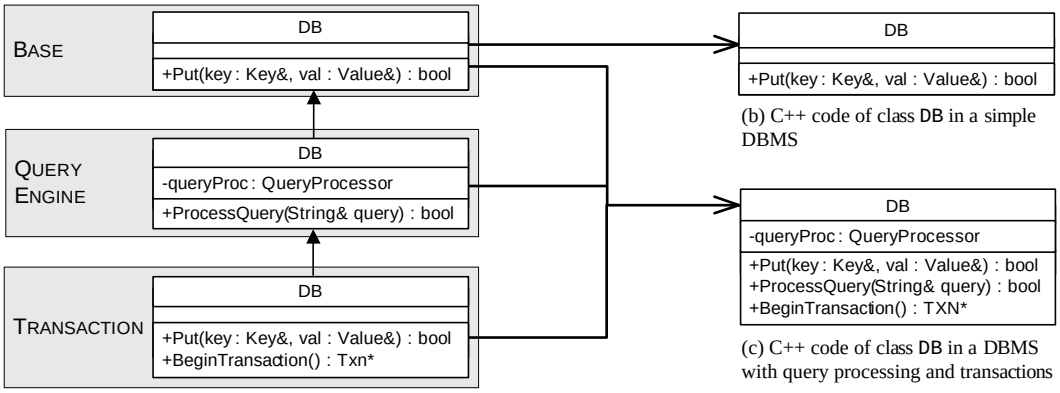
Figure 2.8: FeatureC++ source code of class DB of a DBMS.

tions support composition of different kinds of software artifacts and use a folder of the operating system to aggregate all artifacts of a feature. In the following, we use FeatureC++ to explain the concepts of feature composition for object-oriented classes.

FeatureC++

With *FeatureC++* [ALRS05], we developed an FOP language extension for the C++ programming language [ALRS05, Ros05, ALS06]. An excerpt of the FeatureC++ source code of class DB is shown in Figure 2.8. The basic implementation (Lines 1–3) includes functionality needed for every DBMS variant. It is extended by refinements in features QUERYENGINE and TRANSACTION (Lines 4–17), declared by keyword `refines`. In FeatureC++, refinements can introduce new methods (Lines 6 and 11) and extend existing methods (Line 13). In method extensions the refined method is invoked using the keyword `super` (Line 15). Based on this decomposition of classes and a user's selection of features, classes are composed to include only functionality necessary for a specific DBMS. Code that is not needed is not included in a composed program. For example, the code in Lines 4–9 is not present in a DBMS if feature QUERYENGINE is not selected. As a result of the feature selection process, the code size of class DB is reduced.

FeatureC++ source code is processed by the FeatureC++ precompiler. It uses a source-to-source transformation from FeatureC++ code into C++ code which is then compiled by an ordinary C++ compiler. In Figure 2.9, we show an example for the code transformation of class DB (cf. Fig. 2.8) using UML. Depending on the fea-



(a) FeatureC++ code of class DB

(b) C++ code of class DB in a simple DBMS

(c) C++ code of class DB in a DBMS with query processing and transactions

Figure 2.9: FeatureC++ code transformation: UML representation of class DB in FeatureC++ (a) and two different variants of the composed class DB in C++ (b and c).

ture selection, different variants of class DB can be generated. For example, a simple variant (Figure 2.9 b) is derived by using only the implementation defined in module BASE and a full variant, including features QUERYENGINE and TRANSACTION, is derived by composing all refinements into a single class (Figure 2.9 c).

Virtual Classes

Virtual classes [MMP89] can also be used to implement the features of an SPL. As supported by CaesarJ[9] [AGMO06], the implementation of an SPL with virtual classes is based on nested classes and mixin composition according to the mixin layers approach. A virtual class is a nested class whose type depends on the type of an object of its enclosing class. As in approaches for static mixin layer composition, the enclosing class represents a feature and inner classes represent base classes or refinements of an SPL. With mixin-based inheritance, an enclosing CaesarJ class composes multiple base classes including their inner virtual classes. The inner classes inherit from the inner classes of the parent layers as in static mixin composition. Similar to FeatureC++, CaesarJ allows a programmer to implement the inner virtual classes in separate files. The corresponding enclosing class (which corresponds to a feature) only has to be referenced similar to a Java package in the beginning of the file. This simplifies implementation of large features that contain many classes. Other approaches that support virtual classes are *Delegation Layers* [Ost02] and *Object Teams* [Her02].

[9] http://caesarj.org

Implementing Feature Interactions with FOP

Features often depend on the functionality provided by other features. They may also interact with each other, which can result in unexpected behavior of a program [CKMRM03]. These *feature interactions* can often not be avoided but have to be handled in some way at the implementation level.

As an example for interaction of two features, consider the DBMS from Figure 2.7. Features QueryEngine and Transaction can be used independently, but they interact if using both: the query engine has to parse transaction queries and prepare creation of a transaction. Obviously, this interaction is required to support processing of transaction queries. When both features are used separately, the code for processing transaction queries is not needed. When the features are used in combination special interaction code, also called a *derivative* between the features, is needed [LBL06a]. Hence, a developer may create a special module to implement the derivative code in the same way as a usual feature module. When composing a program, the derivative is only included in the program when both features are included as well.

The example above illustrates interaction between two features. However, there may also be interactions between more than two features which results in *higher order derivatives* [LBL06a]. That is, the derivative is only required when three or more features are used at the same time.

2.3 Feature Binding Time

In this section, we provide an overview of feature binding times important for SPL development. We define static and dynamic binding as we use them in the context of this thesis and present mechanisms to implement different binding times.

2.3.1 Binding Time

The term *binding time* is used in programming languages and SPL engineering in a similar way. In object-oriented programming, we distinguish between *early binding* and *late binding* of methods. Late binding is used when the compiler cannot derive the exact type of an object, which results in binding of method calls at load-time or runtime. Similarly, a feature can be bound at different points in time. When it is known before program start, which features are required in a program, static binding can be used. Dynamic binding must be used when this is not known.

There are different possibilities to categorize the binding time of features in SPLs. For example, Kang et al. distinguish between binding at *compile-time*, *load-time*, and *runtime* [KCH+90]. According to their definition, load-time binding means that the feature binding does not change after binding, whereas it may change afterwards when using runtime binding.

As shown in Figure 2.10, we can distinguish even more binding times. For example, we can differentiate between binding in a preprocessing step (i.e., before

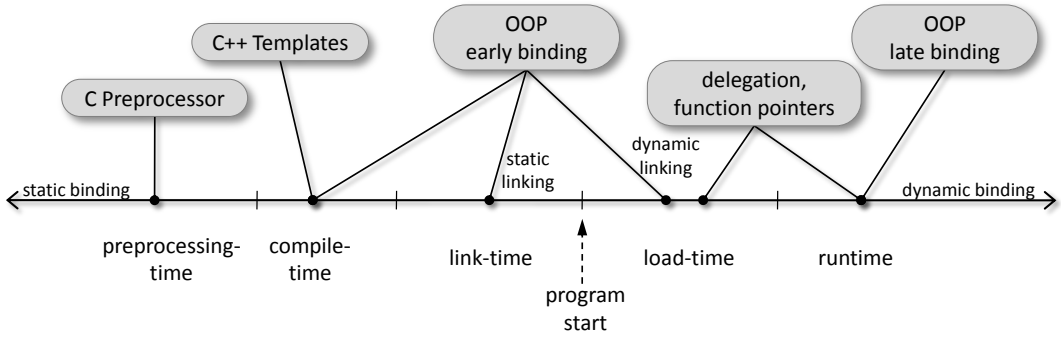


Figure 2.10: Binding times of selected mechanism used in software development.

compilation), binding at link-time (i.e., after compilation but before load-time), or at other phases of the software life cycle [ABE+97]. The different binding times may be important to a particular SPL, application scenario, or user requirements. Simos et al. generalized these different possibilities for feature binding using the term *binding site* [SCK+96]. A binding site represents the point in the life cycle of an SPL at which a feature is bound plus additional binding context.

Similar to Lee et al. [LM06], we distinguish between *static* and *dynamic* feature binding. The distinction between static and dynamic binding has the most far-reaching consequences, e.g., with respect to resource consumption. Furthermore, different kinds of optimizations are possible when static or dynamic binding is used [ABE+97]. We thus use the following definitions:

Static Feature Binding: Static binding means that features are bound before program execution (e.g., in a preprocessing step).

Dynamic Feature Binding: Dynamic binding means binding in a starting (load-time) or a running program (runtime) and can depend on the dynamic context.

In Chapter 5 and 6, we provide basic mechanisms to support dynamic binding in general. However, we focus on binding at load-time, which requires a subset of the needed binding techniques. In Chapter 7, we present extensions required for feature binding at runtime.

2.3.2 Implementing Different Binding Times

Concepts of the underlying SPL development techniques and software composition techniques (e.g., the C preprocessor or components) are used to realize static and dynamic binding of features (cf. Figure 2.10). These concepts also have an influence on non-functional properties like runtime behavior or memory consumption of an SPL. In the following, we describe how to realize different binding times with the C preprocessor, procedural programming, and object-oriented programming. We

focus on compiled programming languages and do not consider dynamic languages that usually support dynamic binding only.

In Figure 2.10, we summarize the implementation mechanisms and the binding times they support. We selected binding times that are important for the implementation techniques and SPL development. In practice, there may be even more binding times and the actually supported binding time depends on several factors, such as the programming language and the execution environment. Everything shown on the left of *program start* in Figure 2.10 is static binding; dynamic binding is shown on the right side of the figure. In the following, we describe binding details specific to the mechanisms that can be used for implementing SPLs.

The C Preprocessor

The C preprocessor supports static binding by removing annotated code fragments (cf. Sec. 2.2.2). In contrast to approaches that support dynamic binding, preprocessing, and thus the feature binding, occurs before compilation and does not hinder further optimizations by the compiler and the linker. The C preprocessor supports static customization with `#ifdef` statements and macros:

- Macros allow a programmer to define code that is directly inlined (called macro expansion) instead of calling a regular method. It allows to avoid repeating the implementation of all kinds of code elements in different places (e.g., defining the same variables multiple times) and thus allows to reduce code replication.
- Code annotated with `#ifdef` statements is only included in a program if actually needed.

Both concepts can be combined by annotating different macro definitions with #ifdef statements, as described in Section 2.2.3.

Since preprocessing occurs before compilation, the C preprocessor enables a programmer to implement variability without loss of performance. This even applies to crosscutting features like a transaction management system of a DBMS. However, macro expansion may also result in code replication: macros that are expanded in several places may result in a copy of the same code fragment. For large amounts of repeated code there may be a significant increase in binary size. In contrast, inlining of a compiler is usually based on heuristics to decide whether to inline a code fragment or not. Repeated inlining does not only increase the binary size of a program. If the size of the executable code is too large to fit into the cache of a CPU, it may degrade performance because the code has to be frequently loaded from working memory. Hence, macro expansion may also increase memory consumption and degrade performance.

Procedural Programming

Procedural programming aims at structuring the source code into procedures or functions that perform a particular task such as an algorithm [Str94]. In com-

piled programming languages, function calls are bound at different stages either at compile-time (by the compiler), at link-time (by the linker), or at runtime (by the runtime environment; cf. Fig. 2.10). In the following, we describe binding mechanism available in compiled programming languages that support procedural programming such as C/C++ and Pascal.

Compilation. After preprocessing steps, a program is *compiled* and *statically linked* to combine binary code units. At compile-time, the compiler resolves function calls within a compilation unit. That is, all function calls that refer to a function definition in the same compilation unit are either inlined or a function call in binary code is generated. For functions that are defined in a different compilation unit, the compiler generates placeholders for function invocations that are filled by the linker.

Static Linking. After compilation, a linker combines the object files generated by the compiler. At this time, function calls across object files are resolved. For functions that are defined in *dynamically linked libraries (DLLs)*, the linker generates a table of imported functions. At runtime, the table stores the addresses of dynamically imported functions. Function calls that refer to such a function are statically bound to the corresponding entry in dynamic function table.

Dynamic Linking. The table that stores dynamically imported functions is filled when loading a DLL. Due to the indirection of the import address table, calls to functions imported from a dynamically linked library are usually more expensive than a direct function call.

Function pointers. Procedural programming, as supported by the C and C++ languages, provides means to store the address of methods in function pointers. This allows a programmer to define and change the binding of a method at load-time and runtime (cf. Fig. 2.10). A function pointer stores the address of a method and can be changed at any time a program is running. Calling a function via a function pointer means to load the address stored in the pointer and continue execution at this address. Hence, it is more expensive than a usual function call, which does not require loading the address separately. Binding function pointers at load-time as shown in Figure 2.10 means that function pointers may be initialized when loading a module and do not change later.

Generic Programming

Generic programming allows programmers to abstract from a concrete type used in a computation by providing a generic type (or value), e.g., as parameter of a method or a class [CE00]. An example is the C++ template mechanism. The generic parameters may also be complex types or functions, which allow programmers to use generic programming for component composition of single classes [VN96b] or

sets of classes [SB98]. In C++, the generic parameter of a template is replaced with a concrete type (in case of type parameters) or a concrete value (in case of integral parameters) during *template instantiation* at compile-time. This enables static binding of methods and data.

Object-oriented Programming

OOP is often used to implement components and is important for the binding time that can be achieved when composing components. In object-oriented languages, objects interact via messages. In compiled object-oriented languages the exact type of a message receiver may already be known during compilation or may only be available runtime. This requires using *early binding* or *late binding* respectively. These different binding types are important for developing SPL with components and have benefits and drawbacks [AG01].

Early Binding. Early binding relies on the static type of an object that can be determined at compile-time. However, in languages such as C++, the concrete implementation of an object's method does not have to be available during compilation. It can also be provided later during linking. Hence, early binding uses the same mechanisms we described for binding at compile-time and link-time in procedural programming. In C++, it depends on the location where a method is defined, whether the actual binding is done at compile-time (when stored in the same binding unit) or at link-time (when stored in a different compilation unit or in a DLL; cf. static and dynamic linking above). In Java, linking always occurs dynamically and starts when loading a class file. Early bound methods that are never used may still be contained in a program and thus increase the binary size. This is avoided by optimizations during compilation and static linking. When using late binding, this optimization is usually not possible because the compiler cannot statically decide whether the method will be used or not.

Late Binding. Late binding relies on an object's dynamic type. It is used if the actual type of a message receiver is not known statically and is determined at runtime. This is usually done at object creation time (e.g., in C++) or at method invocation time (e.g., in Java). Binding via name at runtime, as in Java, means that the binding may change afterwards. Programming languages, like C++, that support receiver identification at object creation time store the result until method invocation. In C++, this is done by storing a pointer to a *virtual function table* (containing all virtual methods of a class) within the created object [Lip96]. This results in additional memory that is needed to store the type of the object implicitly as pointer to the virtual function table. Nevertheless, it increases flexibility by resolving the method that is called depending on the dynamic type of a message receiver.

Delegation. Similar to the use of function pointers, *delegation* [Lie86] can be used to decide at runtime which object a particular message should be delegated to. By changing the delegate, a different implementation of the method can be selected. For example, in the *decorator* design pattern the receiver of a method can be changed at runtime by using delegation [GHJV95]. We will use the decorator pattern to implement dynamic binding of features and dynamic composition of classes.

Aspect-oriented Programming

Advice defined via AOP can be bound at any point in time and depends on the programming language and runtime environment. For example, AspectJ supports weaving at compile-time, after compilation, or at load-time [Asp10]. AspectC++ supports weaving before compilation or at runtime [GS05].

2.3.3 Static and Dynamic Binding of Features

The presented mechanisms support different binding times for data and methods in software development in general. They are used to realize different binding times in SPL development. Since different features may require different binding times multiple mechanisms are combined in practice [vGBS01]. For example, early and late binding can be used to implement static and dynamic binding of features respectively. Similarly, other mechanisms can be used as well. For example, if the C preprocessor is used to implement features, the features are statically bound in a preprocessing step. Hence, the features of a single SPL may be bound at different points in times. This not only means static and dynamic binding. In general it is possible to bind parts of variability at arbitrary points in the deployment phase of an SPL (e.g., all using static binding). This is called *staged configuration* [CHE04].

Using a particular mechanism to implement feature binding does not mean that a program of the SPL has to use only this mechanism. For example, the C preprocessor can be used to implement static feature binding while late binding is still needed to support polymorphic use of classes within a concrete program of the SPL.

Hiding the Binding Time

When implementing a component, the actually required binding time may be unknown until deployment time. Hence, it is important to hide the binding time from a components implementation [Aßm03]. This allows the composition process to choose the required binding time. There are techniques for software development that directly hide the binding time by allowing the programmer to choose the binding time at compile-time [CE00, CRE08] (cf. Sec. 3.2). This can also be achieved by non-invasive modifications of a components implementation. For example, C/C++ developers use binding-time-specific macros to define how methods of a library are linked. This allows for changing the binding time of a library from static to dynamic linking by choosing the appropriate macro definition at compile-time.

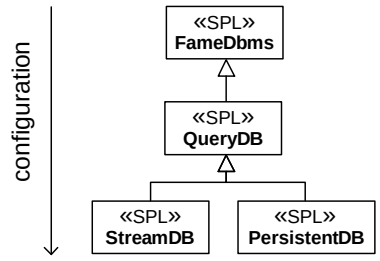


Figure 2.11: Specialization hierarchy of FAME-DBMS.

2.3.4 Staged Configuration

Product configuration (and thus feature binding) can occur in a step-wise manner, which is called *staged configuration* or *SPL specialization* [CHE04]. Staged configuration eliminates configuration choices and thus reduces the variability of an SPL. Hence, a specialization does not specify an SPL instance completely; it is only a partial configuration that still provides some variability. A specialization step does not necessarily bind features of an SPL. For example, it may explicitly exclude a feature. In general, arbitrary constraints can be used to create a specialized SPL by reducing the number of valid configurations [CHH09, RS10].

A *fully specialized* SPL represents a single configuration only [CHE04]. It is the result of one or more specialization steps. We can use a fully specialized SPL directly to derive an SPL instance (e.g., by composing components that correspond to the configuration). In contrast, when creating an instance from an incompletely specialized SPL, we have to bind remaining variability in a final configuration step. That is, we have to fully specialize the SPL first. This can be done in an interactive configuration process or by discarding all features that are not included in the specialization. Discarding remaining features is only possible when it results in a valid configuration. For example, deriving a product from a DBMS specialization that includes a variant feature with alternative children requires choosing one of the alternatives.

Staged configuration means that variability is bound at different points in time and often also at different binding sites but finally results in a complete configuration that defines which features are required for product generation. However, even the final product may be *reconfigured* which results in an infinite configuration process [CHH09].

SPL Specialization Hierarchies

As we described in [RSK10], we can represent SPL specialization steps as inheritance between SPL specializations. As a simple example, consider the specializations of FAME-DBMS shown in Figure 2.11. Each specialization is a partial configuration that inherits from a less specialized SPL. Specialization of FAME-DBMS thus results in a *specialization hierarchy*. Each child in the hierarchy is a specialization of

its parent (i.e., it represents less products), which we express by adding arbitrary *configuration constraints* [RSTS11]. For example, QueryDB is a specialization of FameDbms including a feature for query processing. We can represent the feature selection by adding a *requires* constraint for the feature. As a result, the feature is included in all products that can be derived from QueryDB. StreamDB and PersistentDB further specialize QueryDB. A specialization step does not necessarily add features to a configuration. For example, it may also explicitly exclude a feature. In general, arbitrary constraints can be used to create a specialized SPL by reducing the number of valid configurations. We can also define subtype relationships between SPLs according to the defined hierarchy, as we describe in Section 4.3.1.

In the context of static and dynamic binding, a specialization hierarchy may represent static or dynamic configuration steps. For example, QueryDB may be derived by statically selecting feature QueryEngine. The concrete DBMS StreamDB and PersistentDB may be configured dynamically by adding features Stream and Persistent at load-time.

3 Towards Flexible Binding Times

Binding time flexibility means that we are able to choose the binding time of a feature per application scenario. Since FOP considers features as elements for composition, it provides a proper foundation to support binding different binding times for individual features. We thus use FOP and apply a flexible feature composition process to abstract from details specific to the binding time.

In the following, we analyze properties of static and dynamic binding such as resource consumption and flexibility. We review existing approaches to implement variability with respect to supported binding times. We especially focus on support for static and dynamic binding of entire features, which is important for applicability of the techniques for SPL development. Based on this analysis, we formulate the goals for the thesis.

3.1 A Comparison of Static and Dynamic Binding

Static and dynamic binding are opposite approaches with respect to several aspects of software development (e.g., implementation effort) and properties of program execution (e.g., flexibility and non-functional properties of a program). In the following, we analyze the properties of static and dynamic binding in more detail and show that a combination of both approaches is beneficial for several reasons and sometimes even required.

3.1.1 Binding Time Flexibility

Most SPL development approaches require programmers to choose between static and dynamic binding. However, both binding times provide benefits and it is not always possible to use a single binding time only.

Static binding (e.g., using the C preprocessor) enables optimizations of program code that are hard to achieve at runtime [ABE+97]. Examples are program transformations (e.g., in the Boost library[1]), code generation, or compiler and linker optimizations [CE00]. Applying arbitrary transformations to a running program requires code replacements at runtime. That is, program code is replaced while it is executed, which is usually a difficult task.[2] Hence, static binding provides high flex-

[1] http://www.boost.org/

[2] In general, it is also possible to transform code during runtime, but achieving consistency of transformations (e.g., consistent adaptation of state) is not always possible because the transformation may require information that is already lost during program execution. For example, we cannot correctly initialize an introduced variable that stores the uptime of a program when

ibility with respect to the transformations that can be applied to a program. However, static binding does not allow us to include information in the transformation process that depends on the context at runtime, which in turn reduces optimization opportunities [ABE+97].

Dynamic binding, on the other hand, does usually not support arbitrary program changes but provides flexibility after program start and allows a user or a program to select the needed functionality at load-time or runtime [ASB+09]. For example, loading only required functionality on a mobile device from a network according to the underlying hardware or user preferences can reduce network load. This kind of flexibility cannot be achieved with static binding.

Using static or dynamic binding exclusively is not feasible, because different features may require different binding times [vGBS01]. Static binding can often not be used if required features are not known at deployment time or if they are developed independently, as it is the case for third-party extensions. It is also not feasible to use dynamic binding only. Platform- and compiler-specific features for instance have to be selected before compilation of a program and some devices (e.g., deeply embedded systems) do not support dynamic binding when it requires to load new code at runtime. Consequently, it is sometimes required to support static and dynamic binding at the same time. For example, when a dynamically bound feature is crosscut by a platform-specific feature, we have to statically bind the platform-specific feature, which customizes the dynamically bound feature. Additionally, dynamic binding may increase the resource consumption and it usually means a higher implementation effort to support dynamic binding of features. It is thus often not desirable to bind a feature dynamically [Sim95]. Furthermore, there may be dependencies between features with respect to their binding time. For example, when a dynamically bound feature requires another feature of an alternative group, the group must be bound dynamically as well. Finally, flexible binding times improve reuse of features when different application scenarios or runtime environments require different binding times [ASB+09]. For example, the same feature of a DBMS may be bound dynamically on an embedded device with very limited resources and statically on an embedded device with less resource restrictions.

3.1.2 Compositional and Functional Overhead

A main difference between programs that use static and dynamic binding is the resulting resource consumption of a program (e.g., binary program size, consumed working memory consumption, execution time). Both binding times may increase the resource consumption of a program for different reasons. We distinguish between *functional* and *compositional* overhead:

Functional overhead: *Static binding* causes a functional overhead due to features that are not used but present in a variant.

this information is only available at program start. The result may be an inconsistent state of the whole program. It furthermore requires a high effort to achieve consistency, which can make highly invasive dynamic modifications impractical.

Compositional overhead: *Dynamic binding* introduces a compositional overhead due to additional code for binding features at runtime.

We observe a *functional overhead* when only a subset of the features of a program is used. Remaining features may only be used for a short period or they are not used at all, which depends on the context at runtime. This results in increased binary size and working memory (due to unused code) but also in increased execution time, e.g., due to execution of initialization code. For example, the required functionality of a DBMS, deployed on a smartphone or PDA, depends on the requirements of the applications that use the DBMS. A Web browser that stores encrypted passwords in a database requires a DBMS with a data encryption feature. If the Web browser is never used, the encryption feature of the DBMS is also never used. Hence, the functional overhead depends on the actually used features; it may even change when we run the same program several times, which makes it hard to predict.

We observe a high *compositional overhead* with respect to performance and footprint when functionality is dynamically bound [Big98, Gri00, CE00, HC02, CRE08]. For example, when a method is implemented in several features, the correct implementation has to be chosen at runtime, which requires execution of additional code (e.g., using a conditional statement). This additional code increases the binary size and the execution time of a method because it is executed every time the method is called. Depending on the chosen implementation mechanism (e.g., using function pointers), dynamic binding may also hinder method inlining and increase memory consumption. This is usually needed when the dynamically bound code is loaded at runtime.

Hence, there is a trade-off between static and dynamic binding with respect to performance, memory consumption, availability of bound modules, etc. Current development techniques force developers to choose the lesser evil or she may mix both binding times. The latter results in different implementation mechanisms for different features within an SPL (e.g., preprocessors, design patterns, etc.), which hinders reuse of a feature. Usually, the developer has to choose the binding time per feature before development [LK06]. This decision is crucial since it cannot be easily changed once the feature has been implemented.

3.1.3 Benefits and Drawbacks Summarized

We summarize benefits and drawbacks of static and dynamic binding in Table 3.1. The table provides a relative comparison of both binding times with respect to several properties important for SPL development and does not show absolute values. In the following, we describe the results in more detail.

Safety: Static type checking is possible for a whole SPL independent of the binding time [AKGL10, KATS11]. However, semantic errors may occur for a particular feature combination in a dynamically composed variant. Such errors can be detected by testing a fixed statically generated program that does not change. It is also possible to test dynamically composed program variants but

	Property	Static Binding	Dynamic Binding
Safety	static type safety and testing	●	◐
	avoid runtime errors due to composition	●	○
Development	less complex implementation	●	◐
	simple debugging	●	○
Flexibility	arbitrary transformations	●	○
	separate deployment	◐	●
	loading on demand (e.g., from network)	○	●
	simple reconfiguration	○	●
Resources	static optimization	●	○
	dynamic optimization	○	●
	avoid compositional overhead	●	○
	avoid functional overhead	○	●

Table 3.1: Comparison of static and dynamic binding with respect to properties important in software engineering. We denote support for a property with ● (good support), ◐ (partial support), or ○ (poor/no support)

an exponential large number of configurations dramatically increases the test effort and may result in execution of untested combinations of modules. Furthermore, dynamic binding may fail, e.g., because of missing or incompatible modules (a.k.a. *DLL hell* [And00]).

Development: When using dynamic binding at runtime, a programmer has to consider consistency of updates, which complicates design and implementation of a feature. This is usually not a problem when using binding before runtime (i.e., dynamic binding at load-time or static binding). The overhead for dynamic binding may also force a developer to change the design of an SPL (e.g., to reduce dynamic extensions). Finally, dynamic binding complicates debugging if the configuration of a program is determined during program start or even during runtime.

Flexibility: As already discussed, static binding supports arbitrary code transformations after composition, whereas dynamic binding improves flexibility because code units can be deployed separately (e.g., also by a third party), can be retrieved when needed, and can be easily exchanged (even at runtime).

Resources: Static binding not only avoids a compositional overhead but may also reduce resource consumption due to static optimizations (e.g., domain specific transformations; cf. flexibility). Similarly, dynamic binding avoids a functional overhead and allows for optimizations at runtime (e.g., by replacing a component with one that consumes less energy [SRA10]).

		Binding Time[a]		Feature Composition	
		P C L	I R	Static	Dynamic
A	**Static binding**				
A.1	C Preprocessor	●○○	○○	●	n/a
A.2	FeatureC++	●○○	○○	●	n/a
A.3	Jak	●○○	○○	●	n/a
A.4	C++ Mixin Layers	○●○	○○	●	n/a
B	**Dynamic binding**				
B.1	Delegation Layers	○○○	●●	n/a	◐
B.2	Object Teams	○○○	●●	n/a	●
C	**Combined approaches**				
C.1	AspectC++	●○○	○●	●	◐
C.2	AspectJ	○●●	●○	●	◐
C.3	AspectJ & HotWave	○●●	●●	●	◐
C.4	Components	○○●	●●	●	◐
C.5	C++ Templates & Delegation	○●○	○●	●	◐
C.6	Edicts	○○●	○●	●	◐
C.7	Jak & JavAdaptor	●○○	○●	●	●
D	**Mixed approaches**				
D.1	CaesarJ	○●○	○●	●	◐
D.2	C Preprocessor & Components	●○●	●●	●	◐

[a] Abbreviations for binding times: P: pre-compile-time, C: compile-time, L: link-time/post-compilation (including bytecode weaving), I: initialization-time / load-time, R: runtime

Table 3.2: Comparison of approaches for software customization with respect to binding time and support for composition of features. Supported, partially supported, unsupported properties shown as ●, ◐, ○.

Obviously, both binding times have several benefits and drawbacks. This results in different approaches for software development that support either a single binding time or combine both binding times.

3.2 Approaches for Static and Dynamic Binding

In Table 3.2, we give an overview of implementation mechanisms that support either static or dynamic binding of code units, support different binding times for the same code unit (combined approaches), or support a mixture of both binding times (mixed approaches). In the following, we describe all approaches.

A: Static Binding

There are several approaches that support static binding and can be used for SPL development. In the following, we have chosen a selection of representative approaches to illustrate the underlying concepts.

A.1: C Preprocessor. Preprocessors show us how to achieve customizability and performance at the same time. In contrast to many other approaches, the C preprocessor supports customization even within methods by annotating single statements or even parts thereof. Features can be implemented using code annotations. Dependencies between features may be encoded as propositional formulas at the level of conditional preprocessor statements.

A.2: FeatureC++, A.3: Jak. Both languages support FOP and use static binding in a preprocessing step before compilation. Similar to the C Preprocessor, it is thus possible to generate and deploy the source code of a program variant. However, independent development and compilation of single features is problematic because the whole program is required for compilation.

A.4: C++ Mixin Layers. The C++ mixin layers approach [SB98] uses mixin composition of layers (using C++ templates) and their inner classes (cf. Sec. 2.2.5). The configuration takes place at template instantiation time (i.e., at compile-time). In contrast to FeatureC++, CaesarJ, and other FOP approaches, it is complicated and error-prone to implement features with this approach. For example, to solve the *self-problem* [Lie86] (e.g., to consistently instantiate classes with the new operator), not only the parent layer has to be parameterized but also the *final* layer that refers to the actual SPL instance [Ost02]. The layer must be used to specify the type of a class for instantiation; otherwise it results in inconsistencies.

B: Dynamic Binding

In the following, we list approaches that support dynamic binding only. There are also approaches that support static binding as well, which we list as *combined approaches* below.

B.1: Delegation Layers. The Delegation Layers approach allows for composition of layers at runtime [Ost02]. It combines virtual classes (cf. Section 2.2.5) with delegation-based composition. A layer is implemented as a class and can be used to represent a feature. Nested virtual classes correspond to FOP's class refinements and are composed at runtime by composing their enclosing layers. The approach does not explicitly support removal or deactivation of features at runtime but it could be implemented on top of it. Currently, there is no implementation of the approach.

B.2: Object Teams. Object Teams is very similar to the Delegation Layers approach [Her02]. It supports virtual classes and dynamic composition of teams

at load-time or runtime. A team can be used to implement a feature of an SPL, which encapsulates multiple classes [HMPS07]. Furthermore, Object Teams provides AOP concepts using *callin* bindings, which can be used to implement crosscutting features.

C: Combined Approaches

There are some approaches that hide the binding time from the programmer and allow her to choose it after development. That is, they support static as well as dynamic binding of the same code unit.

C.1: AspectC++. AspectC++ is based on *aspect-oriented programming (AOP)*, which aims at modularizing crosscutting concerns in *aspects* [KLM+97]. AspectC++ supports static and dynamic weaving of aspects [GS05, TLSS10]. Static binding is achieved by a source code transformation before compilation. Dynamic binding at runtime requires preparation of the join points before compilation. AspectC++ allows a programmer to implement an aspect once and decide later whether static or dynamic binding should be used.

C.2: AspectJ. AspectJ is an AOP approach for Java that supports static bytecode weaving at compile-time, after compilation (*post-compile-time*; corresponds to link-time in Table 3.2) and dynamic weaving at load-time [HH04, Asp10].

C.3: AspectJ & HotWave. AspectJ can be combined with HotWave, one of the most advanced AOP approaches for dynamic weaving of aspects [VBAM09, WBA+10]. The combination enables weaving of the same aspect before runtime (using AspectJ) or into a running program (using HotWave). Other AOP approaches that can be combined with AspectJ and support dynamic weaving are SteamLoom [BHMO04] and PROSE [NAR08].

C.4: Components. The binding time in component-based approaches highly depends on the implementation mechanism. Components implemented with C++ support dynamic binding using dynamic linking and late binding of OOP [Str94]. Furthermore, programmers can use the C preprocessor to switch between static and dynamic linking by replacing the import and export declarations of a component.[3] However, statically linking a C++ component does not mean that late binding is replaced by early binding because calls to virtual methods cannot be replaced by usual method calls. Hence, a statically bound component still introduces a compositional overhead when implemented with late binding.

C.5: C++ Templates & Delegation. An approach by Eisenecker at al. implements features according to the GenVoca approach (cf. Sec. 2.2.5) using static binding of C++ mixins and delegation-based dynamic binding [EC00]. By

[3] This has to be distinguished from a mixed approach (cf. approach *C Preprocessor & Components*), which extends components by using the C preprocessor to customize a component before compilation.

combining C++ templates with delegation it is possible to switch the binding time of a feature before compilation. A C++ template metaprogram provides a composition mechanism that allows for checking validity of a static feature selection. The approach may be combined with C++ Mixin Layers [SB98] to enable static composition of entire features.

C.6: Edicts. The Edicts approach uses aspects to support static and dynamic binding by weaving different design patterns using AOP [CRE08]. By switching the aspects, the binding time of a code module can be changed.

C.7: Jak and JavAdaptor. JavAdaptor aims at runtime adaptation with the focus on reducing the compositional overhead and enabling arbitrary changes at runtime [PKG+09, PKS08]. It supports modifications at runtime by deploying the changes of a modified program. JavAdaptor can be combined with any approach that generates Java bytecode. The combination of Jak and JavAdaptor allows for reconfiguring a running program with respect to features [PSC09]. In contrast to other dynamic approaches, this can be used to support binding of whole features at runtime. Reconfiguration with JavAdaptor means to compile a new program before applying the changes and does not support to freely compose precompiled features.

D: Mixed Approaches

Approaches for static and dynamic binding are already combined in practice. Examples are Mozilla [vGBS01], the Apache Web server[4], or Berkeley DB. In Mozilla and Apache, components are used for dynamic binding (e.g., to support different mail protocols) and the C Preprocessor for static binding. In Berkeley DB, static binding with the C Preprocessor is used for all features but the distributed transaction management feature is bound dynamically using function pointers. All mixed approaches require programmers to decide *already before development* which binding time to use for which code fragments. This is required because they have to implement a code fragment with the technique that corresponds to the chosen binding time.

D.1: CaesarJ. The Java-based language CaesarJ supports mixin-based composition of virtual classes [AGMO06] (cf. Sec. 2.2.5). A CaesarJ class can be used to implement a feature that is statically bound at compile-time. A programmer can spread a virtual base class and its extensions over several files (one per feature), which is very similar to FOP with Jak or FeatureC++. Furthermore, CaesarJ supports dynamic weaving of aspects.

D.2: C Preprocessor & Components. The combination of the C preprocessor and components is interesting because both are mainstream approaches and are already combined in practice. The C preprocessor is usually used for fine-grained customizations of dynamically bound components. In case of many

[4] http://httpd.apache.org/

small extensions, the C preprocessor avoids any compositional overhead. It also supports customization of crosscutting features, which often affect several components [Gri00].

Other Approaches

There are further approaches that allow a programmer to implement an SPL with respect to features such as Aspectual Collaborations [LLO03], CIDE [Käs10], Context-oriented Programming [CHdM06, HCN08], FeatureJ [SP10], Hyper/J [OT00], Java Layers [CL01], Jiazzi [MFH01], Scala [OZ05], and Traits [DNS+06]. These approaches use similar mechanisms to implement variability as the approaches described above. rbFeatures is an annotative approach that supports dynamic feature binding for the dynamic language Ruby [GS11]. In this thesis, we focus only on compiled languages that support static binding.

Support for Feature Composition

Only some of the presented approaches explicitly support feature composition, i.e., composing a set of code fragments that correspond to a feature, which is important for SPL development.

Static Feature Composition. Composition of features to derive a concrete product of an SPL is directly supported by static approaches that are aware of features such as Jak [BSR04] and FeatureC++ [ALRS05]. Similarly, mixin-based composition (e.g., using C++ mixin layers [SB98]) and CaesarJ can also be used to implement the features of an SPL as layers and provide means for feature composition using mechanisms of the host-language (i.e., using mixin-based inheritance). The C preprocessor, can be used to annotate features and to encode feature dependencies in conditional statements. All other approaches provide a way to implement features (e.g., using an aspect in AspectJ) but do not explicitly support feature composition with respect to a feature model. However, tools for SPL development, such as pure::variants [pur04] and Gears [Kru08], can be used to add feature-based configuration to most approaches that support static feature binding. Some of the static approaches provide only limited support for feature development. For example, using an aspect or a mixin layer with nested classes to implement a feature means to implement the whole feature in a single file. Avoiding this, e.g., by using multiple aspects and classes to implement a feature, complicates composition of entire features. The combination of C++ templates and delegation proposed by Eisenecker et al. supports validation of the correctness of a static configuration using a template metaprogram [EC00]. For composition of entire features it could be combined with the mixin layers approach.

Dynamic Feature Composition. Dynamic binding of features requires that the dynamic composition mechanism is aware of features. Such a composition mecha-

nism has to consider crosscutting features, feature interactions, and the validation of correct SPL instantiation. For example, with a dynamic AOP approach we have to manually deploy the classes and aspects a feature consists of and have to handle feature dependencies. The dynamic approaches Object Teams and Delegation Layers provide a dynamic feature composition mechanism. Object Teams provides a *Dependent Activation* pattern that can be used to handle dependencies between features. Delegation Layers support instantiation of whole features but there is no mechanism to handle feature dependencies. Using components, dynamic composition of features may be realized by a component that handles composition of other components according to a feature model. Similarly, JavAdaptor does not directly support feature composition. In combination with Jak and additional tool support it is possible to compose features as well but without support for independent deployment [PSC09]. Mixed approaches, such as CaesarJ and the combination of the C preprocessor with components, require developers to choose the binding time per feature before development. This does not allow for changing the binding time of an already implemented feature.

All approaches that provide binding time flexibility (i.e., C.1-C.7 in Table 3.2) support dynamic binding of single features. That is, all dynamically bound features are bound separately. To avoid this, multiple features may be implemented in a single module (e.g., in a component). This enables dynamic binding of all contained features at the same time, which in turn enables static optimizations of the dynamic modules. It also reduces the runtime overhead and simplifies dynamic binding. However, this method does not support configuration of single features. Hence, programmers have to choose at SPL development-time between high customizability and improved performance.

Summary

Our analysis shows that an integration of static and dynamic binding is needed and already used in software development. Static binding avoids a compositional overhead and enables optimizations at the level of the source code (e.g., function inlining). However, a static approach is not as flexible as a dynamic approach since the functionality of a software product has to be known before deployment. By contrast, dynamic composition introduces a compositional overhead and can be used to reduce the functional overhead. Hence, both approaches are useful for different application scenarios but the concrete application scenario may not be known until deployment or even until runtime. We thus conclude that a programmer should be able to choose the composition technique and binding time after SPL development.

Only static and dynamic approaches fully support composition of features and provide a feature composition mechanism. There are several reasons for that. First, feature-oriented SPL development requires a mechanism to implement features including code to resolve features interactions. Second, dynamic feature binding requires a metaprogram that manages feature composition, handles dependencies, and decides which modules to compose at runtime. Third, combining both binding times

means that different composition mechanisms have to be integrated, which has to be considered when dynamically composing features.

Implementation techniques that support static and dynamic binding for different features achieve limited reuse, because source code developed for static composition cannot be easily reused for dynamic composition and vice versa. For example, we can use dynamic composition of features as provided by Delegation Layers and apply a preprocessor, e.g., to support fine-grained static composition of the layers.

Approaches that support static and dynamic binding, support only dynamic binding of individual features when developed as separate modules. Hence, the compositional overhead for binding many features dynamically is very high. In CBSE the overhead is commonly reduced by combining multiple features into a single component; however, this limits customizability and reuse of components because several features always have to be used in combination.

3.3 Perspective and Goals

The use of current SPL implementation techniques shows that different binding times are required for SPL development. We argue that an approach for SPL development should support binding time flexibility to abstract from different binding times and also from the used binding mechanisms. We propose to use FOP for SPL development across application domains including resource constrained embedded systems. FOP languages explicitly support modularization of features and do not require a particular binding time. With a generative approach for feature composition we demonstrate that FOP is a proper candidate to support binding time flexibility. In contrast to other approaches that support binding time flexibility, we argue that dynamic binding should not mean to bind individual features separately. Component approaches demonstrate that a set of dynamically bound features should be bound at the same time [LK06].

In the next chapters, we show that FOP is a viable solution for SPL development that supports different binding times and also binding time flexibility:

1. We provide an approach for static feature composition that can compete with annotation-based approaches such as the C preprocessor. We evaluate the approach in a case study (Chapter 4).
2. We provide a mechanism that allows us to support also dynamic feature binding based on the same feature-oriented code that is used for static binding (Chapter 5).
3. We present a seamless integration of static and dynamic binding of features that overcomes the limitation of existing approaches (Chapter 6).
4. We combine static binding and reconfiguration at runtime (Chapter 7).

In all chapters, we demonstrate the applicability of the approaches with FeatureC++ and show how an FOP language can be used to support different binding times. In the following, we describe the goals in more detail.

3.3.1 Static Binding of Features

Using static binding, we demonstrate that FOP can also be used in application domains where resource consumption plays an important role. FeatureC++ already supports static binding of feature modules. However, the implementation is not optimized for resource consumption and makes use of late binding to implement method refinements. Furthermore, there is no analysis of the overhead of static binding with FeatureC++ (and FOP in general). Hence, we aim at improving the FeatureC++ code transformation process and analyze the compositional and functional overhead caused by static binding. We compare FeatureC++ with customization with the C preprocessor, which is the default tool for customization in the embedded domain. Static feature composition results in a variable interface of the generated program. Hence, we have to ask how a client program can use a statically generated component. We thus have to analyze problems due to interface variability caused by static binding.

Envisioned results:

- Optimization of the static code transformation process of FeatureC++.
- Comparison of customizability and resource consumption of FeatureC++ with the C Preprocessor.
- Analysis of the functional overhead.
- Analysis of interface variability: How to reuse programs (or components) with a variable interface.

3.3.2 Dynamic Binding of Features

Dynamic binding is currently not supported in FeatureC++. We thus extend the code transformation process to support dynamic feature binding at load-time. We choose binding at load-time because it provides high flexibility but avoids the complexity of feature initialization due to binding at runtime. However, the approach is based on delegation and can be extended to support binding at runtime as we show in Chapter 7. We also provide mechanisms to automate SPL instantiation according to a feature model and to verify correctness of a configuration before creating an instance.

Envisioned results:

- Analysis of requirements on the FeatureC++ language for dynamic feature binding.
- A code transformation process that supports dynamic binding of feature modules implemented with FOP.
- An approach for automated SPL instantiation including validation of configurations.
- Analysis of the compositional overhead of dynamic binding.

3.3.3 Integrating Static and Dynamic Binding

To overcome the limitations of using a single binding time, we statically combine multiple features into larger units for dynamic binding. This allows us to (1) exclude program code that is not required and (2) reduces dynamic binding. To adjust an SPL's feature model to the statically generated binding units with reduced variability, we provide a corresponding feature model transformation. We use the resulting dynamic feature model to ensure correctness of dynamic composition of binding units. That is, we abstract from the binding time also at the model level.

Envisioned results:

- An approach for improved flexibility of feature composition by deciding at deployment time which binding time to use for individual features.
- A code transformation mechanism for statically merging sets of dynamically bound features into dynamic binding units.
- An approach for transforming feature models according to the generation of dynamic binding units.
- Evaluation of the presented approach.

3.3.4 Combining Static Binding and Configuration at Runtime

Based on an integration of static and dynamic binding, we demonstrate the applicability of the approach for binding and reconfiguration at runtime. First, we present concepts for feature-based (self-)configuration, and second, we analyze how static binding can reduce the effort for dynamic reconfiguration.

Envisioned results:

- An approach for reconfiguration at runtime (unload features, apply features to a running program).
- Support for (self-)configuration of SPLs at runtime based on features.
- Evaluation of the approach and analysis of the impact of static binding on the runtime configuration process.

4 Scalable Static Feature Binding

This chapter shares material with the DKE'09 paper "Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB" [RALS09] and the FOSD'10 paper "Improving Reuse of Component Families by Generating Component Hierarchies" [RSK10].

In this chapter, we evaluate FeatureC++ with respect to its applicability for SPL development. We argue that FeatureC++ (and FOP in general) is appropriate to achieve high customizability without a negative impact on performance. This also means that FOP can be applied to application domains with scarce resources such as embedded systems. To demonstrate the applicability of FOP to such domains we compare FeatureC++ and the C preprocessor with respect to resource consumption and customizability.

We first analyze the FeatureC++ composition mechanism and present optimizations that avoid any compositional overhead. We evaluate static binding by refactoring Berkeley DB (which is implemented with the C preprocessor) into a FeatureC++ product line. We demonstrate that we can use FOP to derive programs that are equivalent to a C preprocessor-based implementation with respect to resource consumption. Since Berkeley DB is a DBMS component, we finally analyze the consequences on clients that use variable components developed with FOP.

4.1 Static Binding of Feature Modules

The FeatureC++ compiler uses a source-to-source transformation to support static composition of feature modules. However, the generated code is not optimized with respect to resource consumption. In the following, we present optimizations that are needed to achieve the same performance and memory consumption as it is possible with the C preprocessor, which is the state of the art for SPL development in the domain of embedded systems.

4.1.1 Optimizing Static Composition of Classes

Statically composing the features of an SPL means that the classes of a generated program only have to consist of code of the features selected in the configuration process. That is, each class in a concrete product has to contain only those methods and variables that belong to selected features. First implementations of the FeatureC++ compiler transformed the code of several refinements of a feature-oriented class into

Feature CORE

```
1  class DB {
2    bool Put(Key& key, Value& val) { ... }
3  };
```

Feature QUERYENGINE

```
4  refines class DB {
5    QueryProcessor queryProc;
6    bool ProcessQuery(String& query) { ... }
7  };
```

Feature TRANSACTION

```
8  refines class DB {
9    Txn* BeginTransaction() { ... }
10
11   bool Put(Key& key, Value& val) {
12     ... //transaction specific code
13     return super::Put(key,val);
14   };
15 };
```

Figure 4.1: FeatureC++ source code of class `DB` of a DBMS. The class is decomposed along feature modules CORE, QUERYENGINE, and TRANSACTION.

an inheritance chain [Ros05] as it is also supported by the Jak language [BSR04]. However, this introduces problems with respect to performance and memory consumption. For example, virtual methods have to be used to allow class refinements (implemented as subclasses) to override refined methods. This introduces additional indirections for the method call and may increase the size of objects (for storing a pointer to the virtual function table). To optimize resource consumption, we changed the composition mechanism and compose the code of all refinements from the selected features of a class into a single compound C++ class, also supported by the *jampack* composition tool of the AHEAD tool suite [BSR04]. We thus compose the entire code of the base implementation of a FeatureC++ class and their refinements into one compound C++ class. This class consists of:

- the union of all member variables,
- one method for each method refinement,
- one constructor and destructor for each different constructor / destructor definition, and
- one method for each constructor / destructor refinement.

As an example, consider the source code of our DBMS in Figure 4.1 (cf. Sec. 2.2.5). In Figure 4.2, we depict the generated C++ code that corresponds to the FeatureC++ code of class `DB` when composing the CORE implementation with feature TRANSACTION. This generated code is shown only for illustration and does not have to be read by a programmer that uses FeatureC++. All methods and member variables except the code of feature QUERYENGINE are composed into one C++ class. The base implementation of method `Put` (feature CORE) was renamed to `Put_Core` (Line 3) to provide a unique name for every transformed method. It

```
1  //Core implementation
2  class DB {
3    bool Put_Core(Key& key, Value& val) { ... }
4
5    Txn* BeginTransaction() { ... }
6
7    bool Put(Key& key, Value& val) {
8      ... //Transaction specific code
9      return Put_Core(key,val);
10   };
11 };
```

Figure 4.2: Generated C++ source code of class `DB` using static binding of CORE functionality and feature TRANSACTION.

is called from its refinement in Line 9. Using this kind of transformation, a C++ compiler can easily inline method refinements since they are composed into the same file. For example, method `Put_Core` is inlined by most compilers in method `Put` and does not introduce any overhead for method calls. For the GCC and Microsoft Visual C++ compilers we ensure inlining of methods by generating compiler-specific directives[1]. For other compilers we provide a hint that the method should be inlined. For compilers that do not support inlining and for other programming languages, inlining must be performed by the FOP compiler [KAL07].

Debugging. Jampack composition was initially not the preferred code transformation when generating a program from Jak code [BSR04]. The reason is that it mixes the code of several refinements into a single Java file. When a programmer changes the generated Java files during debugging it is hard to map these changes back to Jak files. This problem does not occur in FeatureC++ because the programmer directly debugs FeatureC++ code and can be even unaware of the intermediate code. This is possible by using the C++ `#line` directive in the code transformation process to tell the compiler and debugger the FeatureC++ source code position (file and line number) that corresponds to a line of generated C++ code.

4.1.2 Case Studies

We used FeatureC++ to develop SPLs and also for refactoring existing code into a feature-oriented SPL. In Table 4.1, we provide an overview of analyzed case studies. In all case studies, we applied the FOP refinement mechanism to decompose object-oriented classes along features. We found that the required variability can mostly be implemented with FOP but sometimes the implementation effort is quite high compared to an annotative approach. We especially observed this when refactoring Berkeley DB from C code into FeatureC++, as we describe in the next Section.

[1] We use `__attribute__((always_inline))` for the GCC and `__forceinline` for the Microsoft Visual C++ compiler. See also http://gcc.gnu.org/onlinedocs/gcc/Inline.html

SPL	Description	KLOC	Features
Berkeley DB	database management system	87.3	35
Fame-DBMS	database management system	14.2	81
FeatureC++	FeatureC++ compiler	20.6	23
NanoMail	e-mail client	6.2	25
SensorNetwork	sensor network simulation	7.3	26

Table 4.1: Size and number of features of software product lines implemented in FeatureC++.

In Fame-DBMS and FeatureC++, we used FOP also to implement variability at the level of languages. That is, we used FOP to implement a family of SQL dialects [SKS+08, RKS+09] for Fame-DBMS and the variability in the FeatureC++ language. In Fame-DBMS, we decomposed the SQL grammar into features and generated a tailor-made grammar according to the features selected for a particular SQL dialect. This requires a special static composition mechanism that merges the rules of a BNF grammar [BLS98, Sun07]. Next, we present a detailed analysis of our Berkeley DB refactoring.

4.2 A Case Study on Berkeley DB

We use Berkeley DB to compare static binding with the C preprocessor with a feature-oriented implementation in FeatureC++. We evaluate benefits and drawbacks of both approaches with respect to resource consumption and aspects of the SPL development process.

4.2.1 Static Binding in Berkeley DB

In Berkeley DB, programmers use the C preprocessor for static configuration. Furthermore, build tools are used to select the files of the code base that have to be preprocessed. This avoids preprocessor-based annotation of whole files. Hence, the used mechanism is actually a combination of a compositional approach (selecting files via build tools such as GNU Make[2]) and an annotative approach (using the C preprocessor).

Besides missing type checking and other drawbacks, some studies point out that the C preprocessor has a negative affect on maintenance of software [SC92]. Because of missing modularization also the evolution of software and even the elimination of dead features is problematic [BM01]. We will only contrast the different concepts to implement variability with the C preprocessor and FOP. A detailed analysis of readability and maintainability is hard to accomplish and is outside the scope of

[2] http://www.gnu.org/software/make/

```
1  __rep_queue_filedone(...) {
2  #ifndef HAVE_QUEUE
3    COMPQUIET(rep, NULL);
4    COMPQUIET(rfp, NULL);
5    return (...);
6  #else
7    db_pgno_t first, last;
8    u_int32_t flags;
9    int empty, ret, t_ret;
10 #ifdef DIAGNOSTIC
11   DB_MSGBUF mb;
12 #endif
13   ... //92 Lines of Code
14 #endif
15 }
```

Figure 4.3: Static configuration in Berkeley DB with nested preprocessor statements.

this thesis. In Figure 4.3, we depict an example of annotations in Berkeley DB code that illustrates why preprocessor statements (especially nesting) may degrade readability, comprehensibility, and maintainability of the source code. Very long code sections between `#ifdef`s (>100 LOC) do often not allow a programmer to easily decide if a part of the source code belongs to a particular feature or another. The preprocessor statements are nested and enclose code that belongs to different features. The intermingled functionality and the size of the functions make local behavior hard to understand.

The customizability achieved in Berkeley DB is limited: some features cannot be removed and others can only be removed partially if code is not entangled with the remaining source code. This is not a general limitation of the C preprocessor. It is caused by crosscutting features which increase the number of nested preprocessor statements when a programmer strives for a complete decomposition of the source code. Separating all features completely and increasing customizability may thus further degrade readability of the source code.

4.2.2 Refactoring Berkeley DB

Most contemporary data management systems are written in the C++ programming language. Because implementations of DBMS are usually highly tuned, they cannot be reimplemented from scratch using a novel programming paradigm like FOP without a huge amount of work and the risk of degrading performance or introducing errors. By refactoring the C version of Berkeley DB[3] into a FeatureC++ product line, we demonstrate that a refactoring approach is appropriate for decomposing legacy DBMS. In a second step, we further decomposed the refactored Berkeley DB version to extract more features, which is also known as *Feature-oriented Refactoring* [LBL06a].

[3]We used the C version 4.4.20 of Berkeley DB.

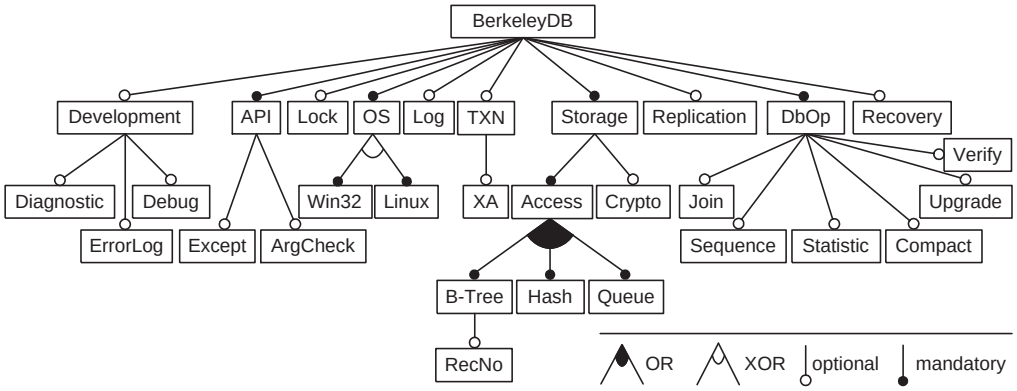


Figure 4.4: Excerpt of the feature diagram of Berkeley DB after refactoring. Only optional and alternative features are shown.

From C to FeatureC++. We used a two step refactoring process: (a) the conversion from C to C++ and (b) the conversion from C++ to FeatureC++. To avoid errors and to preserve the behavior of the software, we used minimal invasive code transformations that do not change the design [Fow99]. By partially automating this process, we are able to transform also large applications into a FeatureC++ product line.

Feature-oriented Refactoring. After transformation to FeatureC++, we extracted further features. This process required manual transformation of the source code to completely modularize the features. For example, crosscutting features often affect large parts of the source code and besides the base program also other features. In Berkeley DB, the transaction management system crosscuts many classes and features of the DBMS (more than 30 classes and 11 features). The B-Tree classes are extended to introduce code for transaction management. Due to the absence of appropriate tool support, we extracted the features manually. That is, we decomposed the classes into base implementation and refinements according to crosscutting features. This means that code encapsulated by `#ifdef` statements is refactored into a class refinement of the according feature, e.g., moving a method into a refinement.

Results. In its original version 11 features are optional and can be individually disabled when generating a concrete instance of Berkeley DB. This results in about one thousand different variants. In Figure 4.4, we depict an extract of the feature diagram of Berkeley DB after refactoring; mainly optional and alternative features. Overall, we extracted 35 (cf. Fig 4.4) features with 24 of them being optional (11 optional features in the original version). The remaining 11 features are mandatory for each variant of Berkeley DB and thus cannot be removed in a concrete configuration. However, we found it still useful to separate these features to increase comprehensibility and to allow for alternative implementations of such features. Based on all

existing constraints between features, there are more than 400,000 valid configurations.

Modularization of Features

Many features that we modularized in Berkeley DB cut across large parts of the base program. These crosscutting features also affect other features, e.g., the transaction management interacts with 11 other features. When extracting such features in Berkeley DB, we made some observations regarding structure and modularity of the source code.

Static Binding. The C version of Berkeley DB already supports static binding of some features using C preprocessor directives. The programmers of Berkeley DB use function pointers to support object-based programming and to exchange the implementation of a method depending on the configuration, i.e., depending on the feature selection. In order to provide this customizability, function pointers are initialized when running Berkeley DB. The according initialization code has to be written manually by the programmer.

When using FeatureC++, object-oriented concepts, i.e., classes with methods, eliminate the need for using function pointers as a mechanism for object-based programming. Furthermore, FOP's composition mechanism replaces the configuration mechanisms used in Berkeley DB: (1) dynamic assignment of functions to function pointers in C is replaced by static selection of the required methods implemented in different features in FeatureC++ and (2) static composition based on the C preprocessor is replaced by different FOP mechanisms, as we describe in the next section. The tool-supported composition mechanisms of FOP thus reduce implementation effort and are less error-prone than a manual setup of function pointers or using the C preprocessor. Hence, FOP introduces safety for composition of variants: a method is always correctly initialized, avoiding runtime errors like accessing a null function pointer, and consistent configuration of all classes is achieved automatically. Furthermore, a type system can assure static type safety of all possible variants of an SPL [AKGL10]. This can in general also be achieved with annotative approaches but requires disciplined annotations [KATS11]. Technical aspects like memory consumption and performance also benefit from using FeatureC++. The reason is that function pointers do not have to be stored in objects and the compiler can inline method refinements, which is not possible for functions referenced by pointers.

Decomposition of Large Methods. Many methods in Berkeley DB are quite large and contain entangled functionality of different features. Reasons for this lack of modularization at the function level are performance optimizations, which can also be achieved by inlining capabilities of modern compilers. For those methods, we created up to seven methods each by using the extract method refactoring [Fow99]. In some cases, we used *hook methods* to decompose large methods. This requires additional effort for decomposing such methods, but tool support could significantly

ease the refactoring of large applications [LAMS05, KTS+09]. Decomposing methods into smaller methods and method extensions, including the use of hook methods, does not have a negative effect on performance due to method inlining.

The effort to modularize a feature depends strongly on its crosscutting nature. Thus when deciding to extract a feature, the effort and the resulting benefit have to be taken into account. In our case study, we did not remove all preprocessor statements because of their high number and the required effort. Hence, tool support on top of FOP is necessary to reduce the refactoring effort especially for extraction of small features [KAK08]. First attempts aim at semi-automated transition of single systems into feature-oriented SPLs [KKB07, Dre10].

4.2.3 A Comparison of C and FeatureC++

Based on the refactoring of Berkeley DB, we compare the variability mechanisms of the C and FeatureC++ implementations with respect to properties important for SPL development.

Variability Mechanisms

In the original Berkeley DB version, several mechanisms are mixed to implement variability:

Code annotations: With conditional statements of the C preprocessor (i.e., `#if`, `#ifdef`), the programmers define optional or alternative code fragments.

Macros: Macros are combined with `#ifdef` statements to define optional code and alternative code fragments.

Build system: The build system (e.g., GNU Make) is used to select files and libraries that have to be compiled for a particular program variant.

Function pointers: Function pointers are used to support dynamic binding and are combined with the C preprocessor to support static binding as well.

In our refactoring, we replaced these mechanisms by FOP concepts. For variability implementation, we use a single concept: FOP's feature modules. We summarized the concepts found in Berkeley DB and their FeatureC++ replacement in Table 4.2. In the following, we compare the concepts and analyze the properties of the variability mechanisms.

Conditional Compilation and the Build System. In Berkeley DB, conditional compilation is used to annotate whole methods or statements within methods (cf. Sec. 2.2.3). Since an entire source file may be part of a single feature, conditional compilation is combined with the build system (Microsoft Visual Studio, GNU Make), which supports compilation of selected files only. That is, the number of code annotations is reduced by compiling a varying set of source files. We could

Variability mechanism	FeatureC++ replacement
annotated method	method + refinements
annotated statements	hook method + refinements
alternative type declaration	alternative `typedef` in features
macro – executable code	hook method + refinements
macro – struct declaration	class refinement
optional & alternative files	files stored in feature modules
function pointers	alternative method implementations

Table 4.2: Replacements for variability mechanisms used in Berkeley DB with FeatureC++ language constructs.

replace both mechanism with FOP using classes and class refinements: annotated methods and statements are modularized into methods and hook methods (described separately below) with refinements; variability encoded in build scripts is replaced by moving source files into the folder of the corresponding feature module. Code that does not correspond to a class (e.g., global variables) is moved into header files without a class that are also composed according to a feature selection.

Macros. C macros are also used to implement variability in Berkeley DB. By defining a macro in a conditional preprocessor block, a programmer can define functionality that is only available if a particular feature is present in a program variant (cf. Sec. 2.2.3). Many macros in Berkeley DB define variables and executable code, which can be replaced by hook methods as we describe below. Furthermore, alternative macro definitions are used to define types that depend on the target operating system. This can be replaced by C++ typedefs in the corresponding features of the FeatureC++ code. Macros are also used to compose C structs according to a feature selection. By defining parts of the body of a struct in a macro, the programmers can include this macro into a struct definition. Combining multiple macros in a struct is very similar to composition of classes using refinements and can thus be replaced by FOP.

Macros vs. Hook Methods. Macros that define executable code can be replaced by hook methods. If there are alternative implementations of a macro (i.e., depending on the feature selection), alternative hook methods have to be defined in different features. In our Berkeley DB refactoring, we also had to create hook methods to decompose methods with `#ifdef`-statements to modularize the code of multiple features. Hook methods are often used in frameworks to introduce extension points into a method (cf. Sec. 2.2.4). In FOP, hooks can be used similarly. They enable subsequent features to introduce specific code into the middle of a method via refining a hook method. For example, in order to provide an extension point for cursor

initialization, we use a hook method `InitNewCursor` that can be overridden by subsequent features to provide initialization code. This hook method is called when creating new cursors objects. Its base implementation is empty and is overridden in features such as B-Tree to execute feature-specific initialization.

Hook methods and their refinements are similar to a combination of macros with conditional preprocessor statements. Both define functionality that depends on the configured features. Comparing hook methods and macros with respect to readability, there is no general difference. The use of hook methods can be complicated if local variables are involved. These variables have to be passed as arguments to hook methods to allow their use in method extensions [KAB07]. In contrast to macros, hook methods can be extended by multiple features when using FeatureC++. Furthermore, hook methods are part of the programming language and the type system. They support type safety and overloading, and are encapsulated in the corresponding class. Additionally, complex `#ifdef` constructs, e.g., nesting, cannot be avoided when using macros since they are still needed for macro definitions. In case of Berkeley DB, 66 % of the source code are part of optional or alternative features and are accessed via `#ifdef` constructs or macros obfuscating most of the source code. Using FeatureC++, we are able to properly modularize such code into feature modules and separate it from other features.

In contrast to hook methods, macros do not have an own scope of execution, which means that a macro is executed as part of the function that contains the macro call. For example, a macro may contain a return statement. Calling the macro thus ends the execution of the calling method. This cannot always be replaced by a hook method and the return statement has to reside outside the hook. Hence, a combination of both concepts may be beneficial in particular situations.

Function Pointers. In Berkeley DB, function pointers are used to achieve an object-based programming style but also for configuration purposes in combination with the C preprocessor and to achieve dynamic variability. The function pointers are initialized depending on a static feature selection. For example, when distributed transactions are used, function pointers for transactions refer to the functions for distributed transactions instead of the regular transactional functions. The use of function pointers for implementing static variability suggests that developers strive for simpler mechanisms for customizability that provide some level of abstraction, i.e., function pointers that can be easily exchanged.

Configuration with function pointers in combination with conditional preprocessor statements is semantically equivalent to method refinements in FOP. For example, code for distributed transactions in Berkeley DB can also be implemented in a method refinement that overrides the original implementation and is only available if distributed transactions are used. Replacing such function pointers with refinements of FOP means an automation of the manual configuration process, i.e., replacing manual pointer initialization by FOP code transformation. A difference between function pointers and FOP is the supported binding time. A function pointer is

usually bound at load-time and may even be rebound at runtime. In Berkeley DB, function pointers are used for static and dynamic binding. Until now, we only considered static binding of FeatureC++. In the next chapter, we present support for dynamic binding.

SPL Development

Fine-grained Customization. FOP supports fine-grained customization by decomposing a method into several method refinements. This, however, sometimes requires to use hook methods when code of a feature has to be introduced into the middle of a method. By contrast, the C preprocessor allows a developer to provide fine-grained customizability by modularizing even single statements. This also means that annotated code remains in the context of the surrounding statements and is not moved to a separate refinement. Especially for fine-grained extensions, annotations may improve comprehensibility of source code by avoiding heavy fragmentation. We also observed this effect in SPLs that we implemented from scratch (e.g., in FAME-DBMS).

Modularization and Reuse. The C preprocessor allows us to achieve high reuse within an SPL without physically separating the features. While modularization often means a higher development effort, it also means better reuse within and also across SPLs. For example, annotated code fragments can only be reused in the context of the surrounding code, i.e, at a specific point within a single SPL. We cannot combine an annotated code fragment with other code that requires the same functionality. By contrast, a feature module can be freely composed with other features (e.g., alternative implementations of a transaction management system). Hence, when adding a new feature to an SPL, it may use functionality of existing features or extend them. This improves reuse within the same SPL. In C and C++ programmers achieve modularization of features by combining the lower level modularization mechanisms that work on files such as `#include` statements, alternative includes, and selection of files by the build system. This also requires additional effort for decomposition of the source code as in FOP. Hence, when modularity and reuse is not required, annotation with the C preprocessor may be preferred over decomposition into feature modules using FOP or the modularization mechanisms of C/C++.

In contrast to these C/C++ mechanisms, an FOP feature module can extend the methods of another feature module (i.e., using method refinements), which is not possible with macros or functions in C/C++. A feature module may even be reused in other SPLs that provide the same extension points, i.e., the points in code required for defining extensions such as method signatures. Hence, the base program that is extended by a feature module must provide the classes with the methods and variables that are extended and used by the feature module. This limits reuse of feature modules between SPLs.

Safe Composition. Using FOP, a programmer can ensure that every generated variant is correct with respect to syntax and semantics of the used programming language, which is known as safe composition [AKL08, AKGL10]. This is only possible with an appropriate type system that is aware of SPLs, which is beyond the scope of this thesis. Modularization of features can improve the comprehensibility of large software systems because not the whole source code has to be inspected to understand a particular feature [KLM+97]. Nevertheless, FOP may also have a negative impact on code comprehensibility, e.g., when hook methods have to be used. For that reason we think that tool support, similar to a *virtual* separation of concerns in CIDE [KAK08], can further improve the comprehensibility of FOP code.

Feature Interactions. Using the C preprocessor, derivative code (i.e., code that must be available only when two or more features are selected) is usually implemented with nested `#ifdefs`. It is thus automatically included in a program when the corresponding features are included. In our FOP refactoring, we transformed this code into a separate derivative module that is included only when the corresponding features have been selected. Similar to the effort for modularizing a feature, it also means a higher effort for extracting a derivative module than annotating the code. However, comprehensibility of the source code may be improved by using FOP as we describe below.

While extracting more features in Berkeley DB, we observed that the number of derivatives increases more than linearly [KAuR+09]. This high increase may be caused by the fact that we refactored an existing application and we did not design it with the aim of reuse. However, the high increase may also be natural for feature interactions and has to be further analyzed. For more details on feature interactions we refer to [KAuR+09, Käs10].

Comprehensibility of Source Code. The comprehensibility and readability of source code depends on several properties and is hard to analyze. For example, separation of concerns is expected to improve the comprehensibility of source code [Dij72a]. Since we cannot directly analyze the impact of the different implementation mechanisms on comprehensibility of the source code, we analyze their impact on separation of concerns and illustrate how this may affect comprehensibility of the source code.

In contrast to components or FOP, the C preprocessor does not directly support separation of concerns in source code. This is partially possible using separately defined macros instead of scattered `#ifdef` statements. The lack of separation of concerns is also the reason for reduced maintainability and comprehensibility of the source code [BM01]. The result is code of features that is entangled with other code and scattered all over the program. Consequently, local behavior can only be understood by inspecting large parts of the code. Comprehensibility is further degraded by nested `#ifdefs` [SC92]. This effect can be reduced by using macros but

cannot be completely avoided because it often results in nesting for macro definitions.

Using FOP, features are modularized and separated from each other. Parts of classes and methods can be related easily to their functionality (i.e., their feature) and can usually be understood without considering code of other features. One of the major features that crosscuts large parts of the source code is the transaction management system (4,208 LOC). In Berkeley DB, we extracted the transaction management system, as other features, and separated it from the remaining functionality. This decomposition decreases the code size of other features. For example, feature B-TREE consists of core functionality and other crosscutting features (cf. Figure 4.6). As a result, one has to inspect only about 50 % of the complete feature to understand the B-TREE. Furthermore, features that crosscut the B-TREE implementation can be easily inspected by browsing the according separated implementations. On the other hand, to understand a particular feature sometimes also the code of other features has to be considered. Nevertheless, there are usually still features that can be ignored. For example, to understand the recovery implementation of the B-tree, a programmer has to understand the features B-TREE and RECOVERY but can ignore all other features that crosscut the B-tree, which make up about 25 % of the code (cf. Fig. 4.6).

FOP can also have a negative effect on readability of the source code when comparing it to techniques that do not support customization. For example, if we use FOP and hook methods to introduce extension points into methods, we may degrade the comprehensibility and do not completely follow the principle of separation of concerns (the method call remains in the original feature). Hence, interactions between features cannot be completely modularized if they occur at the level of statements within methods. In this case, the decomposed source code may be difficult to understand since the program flow switches between a number of methods and refinements. Finally, the degradation of readability is a result of increasing customizability and can also be found in other approaches for SPL development.

Replacing the C Preprocessor

Macros and fine-grained customization capabilities of the C preprocessor demonstrate that it is often hard to replace annotations with FOP. Hence, a combination of annotative approaches with FOP can be beneficial in this case [KA08]. We argue that such a combination is useful for special situations only. Our experience with the Berkeley DB refactoring and development of other SPLs with FeatureC++ shows that there are situations in which we do not want to replace the C preprocessor with FeatureC++ even though it would be possible. For illustration, we give two examples:

- **Preconditions, postconditions, invariants:** Programmers use the C preprocessor to define macros that allow them to check at runtime whether pre- and postconditions and also invariants are satisfied. In some situations, this requires to execute special code when the conditions are not satisfied. Since the

Configuration	Features[a]	Binary Size [KB]	
		C	FeatureC++
(1)	B-Tree, TXN, Others, Queue, Rep, Hash, Crypto	664	636
(2)	B-Tree, TXN, Others, Queue, Rep, Hash	644	620
(3)	B-Tree, TXN, Others, Queue, Rep	580	552
(4)	B-Tree, TXN, Others, Queue	528	484
(5)	B-Tree, TXN, Others	492	452
(6)	B-Tree, TXN	416	376
(7)	B-Tree	N/A	224
(8)	Queue	N/A	184

Table 4.3: Analyzed variants of benchmark applications with Berkeley DB embedded into the program. Shown are the used features and the binary size for C and FeatureC++ variants of the application.

[a]TXN: Transactions (includes Logging and Recovery), Others: other small features, Rep: Replication.

code often includes a `return` statement, it is not possible to replace the code by a method, because the `return` statement must reside outside the method. In some cases this can be replaced by throwing exceptions.

- **Debugging:** It is often required to execute code only for debugging purposes. For example, code for logging and analysis during debug sessions can be complex and is usually highly entangled with other code of a method. The code is often only a method call (e.g., for logging) and a hook method can be used to avoid execution of additional code when building non-debug variants. It may be inappropriate to extract such code into a hook method when the code is highly entangled with the surrounding code and cannot be reused at other places (i.e., it is only used for a single method). Furthermore, a compiler cannot always *optimize away* the arguments of empty hook methods, which means that code required for debugging may also be executed when debugging is disabled.

The pre- and postcondition example can be seen as a way to extend the capabilities of the programming language using the C preprocessor. We could avoid the macros by extending the programming language with user-defined syntax and semantics (e.g., adding pre- and postconditions with particular actions for methods). However, the example can be generalized to features that change the control flow within a method. The debugging example shows that modularization sometimes requires too much implementation effort and may degrade code comprehensibility. This example is a special case of feature code that is specific to a method and whose modularization does not provide a benefit.

4.2.4 Resource Consumption

We analyze the feature-refactored version of Berkeley DB with respect to customizability and resource consumption. We compare several variants with the original C

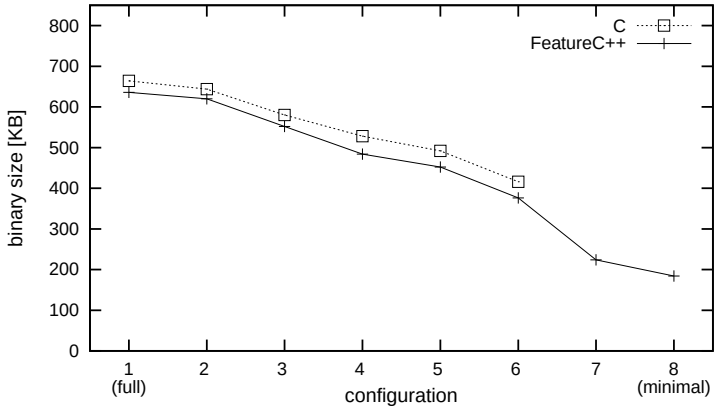


Figure 4.5: Binary size of different C and FeatureC++ variants of Berkeley DB. See Table 4.3 for a description of shown configurations.

version of Berkeley DB.[4] As FeatureC++ uses a code transformation to C++, we want to point out that the use of C++ is sometimes refused because poor performance and high resource consumption is assumed. According to Stroustrup there is no evidence for this argument [Str02], which our evaluation confirms.

For our analysis we use eight different variants of Berkeley DB embedded into a small benchmark application, as shown in Table 4.3. The size of the generated program is decreasing from configuration one to eight. In configurations 1–6, we use the same features in the C and FeatureC++ variants. Configuration 6 is the smallest possible C variant using the index structure B-tree. Configurations 7–8 are minimal variants using B-tree (7) and Queue (8) as index structures. Both variants are not available in the C versions of Berkeley DB because features like TRANSACTION have been removed in these variants, which is not possible in the original version.

Binary Size. The binary size (footprint) of an application depends on the size of executable code and static data. Comparing our feature-refactored version of Berkeley DB with the original version, we observe roughly equal binary sizes for equivalent configurations (i.e., the same set of configured features). Table 4.3 and Figure 4.5 summarize the measured binary sizes of different configurations of the benchmark application. Reasons for a reduction of the footprint in FeatureC++ variants (e.g., from about 664 KB to 636 KB in Configuration 1) are mainly differences in the programming paradigm and code for customization. For example, in the C version of Berkeley DB function pointers are used to mimic object-based programming; these have to be manually initialized when instantiating objects and thus increase the binary size. These differences are negligible compared to the overall size of the application.

[4] The used C and FeatureC++ source code of Berkeley DB is available under http://wwwiti.cs.uni-magdeburg.de/iti_db/BerkeleyDB/

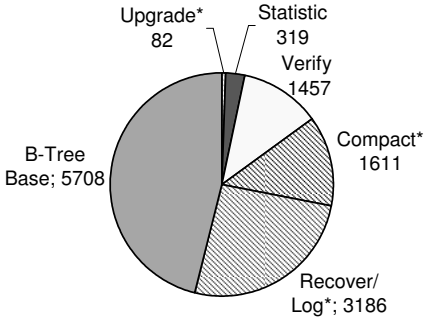


Figure 4.6: Lines of code of feature B-tree consisting of the basic implementation (*Base*) and portions of other crosscutting features.

The binary size of applications that use Berkeley DB can be further decreased by removing features that are not needed (i.e., reducing the functional overhead). When comparing the smallest variants of the feature-oriented refactoring of Berkeley DB (7 and 8) to the smallest C variant (6), we see significant differences (code reduction of about 50 %).

Crosscutting Features. Due to crosscutting features (e.g., RECOVERY), the size of a feature decreases when we extract crosscutting features. For example, the source code of feature B-TREE, as shown in configuration (1), is about 7 KLOC larger when compared to configuration (7), which is caused by removing all features that crosscut the B-TREE. In Figure 4.6, we show the crosscutting features in detail. For example, feature RECOVERY makes up a large part of the B-TREE implementation which is needed for recovery of the index. By omitting feature RECOVERY the size of the B-TREE source code decreases by 3186 LOC.

Performance. As mentioned above, C++ has to be used carefully to avoid performance penalties. We considered this when imposing an object-oriented design on Berkeley DB. For example, we did not use virtual methods. In Figure 4.7, we depict performance comparisons between the C and FeatureC++ variants of Berkeley DB using a reading benchmark.[5] We use the same configurations as used for comparison of the binary size. For configurations 1–6, the performance is roughly equivalent when comparing C with FeatureC++ variants. Configuration 7 is the smallest FeatureC++ configuration using the B-tree index structure.[6] Among others, feature TRANSACTION was removed in this FeatureC++ variant (cf. Table 4.3).

[5] For benchmarking, we used an Intel Core 2 system with 2.4 GHz and operating system Windows XP. For compilation, we used the Microsoft C/C++ compiler v13.10.3077 and Incremental Linker v7.10.3077 (Visual C++ 2003). The used benchmark is a reading benchmark for Berkeley DB available from Oracle: http://www.oracle.com/technology/products/berkeley-db/pdf/berkeley-db-perf.pdf.

[6] Configuration 8 (as shown in Figure 4.5) was omitted since a comparison is not meaningful due to the use of the *Queue* index structure instead of a *B-tree*.

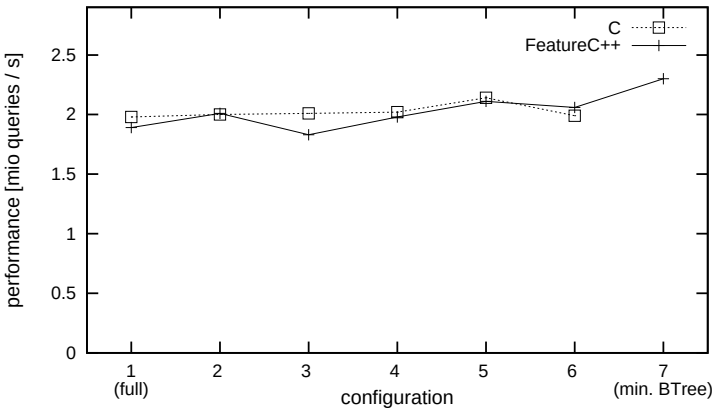


Figure 4.7: Performance comparison (Oracle benchmark) of C and FeatureC++ variants for different feature selections. Higher values mean better performance. See Table 4.3 for a description of shown configurations.

It results in a performance improvement of about 16 % compared to the minimal C version even though the feature is inactive in all variants. This is caused by the removal of dynamic checks that are evaluated even when features like TRANSACTION are not used. Especially for transactions, numerous of such tests are utilized, which explains the better results for configuration 7 in Figure 4.7. These checks are implemented with conditional statements that have to be executed on every database access for some features.

4.2.5 Functional Overhead

In Figure 4.8, we summarize the maximal functional overhead of Berkeley DB variants for an increasing number of features. We depict the relative binary size and the relative execution time of the reading benchmark used in the previous section. The relative overhead means the additional binary size or execution time due to features in a program that are not used, i.e., compared to the smallest configuration. For example, in a full configuration, the binary of the program has a size of 636 KB compared to a binary size of 224 KB for the configuration with 14 features, which means an overhead of 184 %.

The increasing binary size is not surprising; it is caused by the executable code of added features. The increasing execution time, however, is not caused by functionality of added features since we used only basic functionality in all variants when executing the benchmark. For example, even though we added the transaction management feature, we did not use transactions. By contrast, executing additional functionality would heavily degrade performance. For example, using transactions would result in performance below 1 % compared to execution without transactions. This degradation can also be seen as a functional overhead. In case of Berkeley DB, however, the user does not have to use this functionality due to more

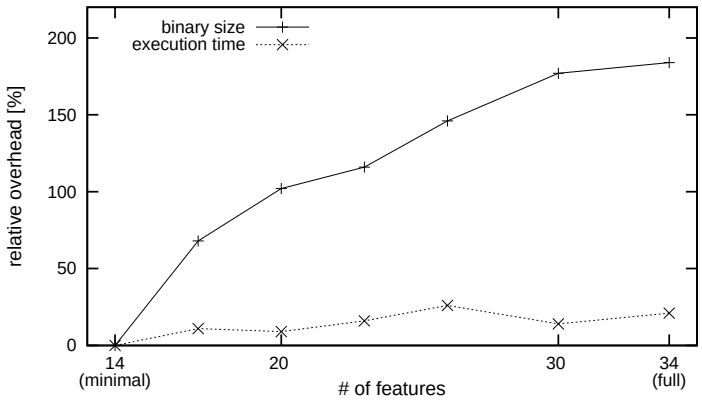


Figure 4.8: Relative functional overhead of Berkeley DB variants with an increasing number of features.

fine-grained possibilities for activation and deactivation of transactions on the basis of single queries. Nevertheless, we observe a slightly increasing execution time when adding more features. The reason is additionally executed code that checks whether a particular functionality is active or not.

4.3 Component Product Lines

Berkeley DB is an embedded DBMS, which means that it is embedded into a program and accessed via an API. A concrete Berkeley DB variant can be seen as a component that is used to build larger software systems (e.g., a stream processing system). The result is *component variability* [vdS04] and we thus call Berkeley DB a *component SPL*. A developer of a client application that uses an instance of Berkeley DB can thus derive a variant that satisfies the needs of a particular application scenario. However, when the requirements change, the generated Berkeley DB has to be reconfigured and rebuilt accordingly. This may result in a varying interface, which complicates client development as we observed it in Berkeley DB.

4.3.1 Product Line Interfaces

In Section 2.2.4, we have seen that components are connected via an interface. This interface is considered to be stable [PBvdL05], which is important for interoperability of components [Par79, Par07]. By contrast, when a component itself is developed as an SPL, the interface may depend on the configuration of the component. In the following, we analyze component variability. We describe reasons for interface variability and define a *semantic SPL interface* and an *SPL programming interface* [RSK10].

Feature CORE

```
1 class DB {
2   bool Put(Key& key, Value& val, TXN& txn) { ... }
3 };
```

Feature TRANSACTION

```
4 refines class DB {
5   bool Put(Key& key, Value& val, TXN& txn) {
6     ... //transaction specific code
7     return super::Put(key,val,txn);
8   };
9 };
```

Figure 4.9: FeatureC++ source code of class `DB` with method `put`. The method refinement in feature TRANSACTION requires argument `txn`.

Interface Variability and Substitutability of Components

The variability of a component's interface causes problems for client development in two situations: first, when changing the configuration of the used component and, second, when polymorphic use of components is required.

Component Reconfiguration. When a component is reconfigured (e.g., adding feature TRANSACTION to Berkeley DB), methods are added to and removed from the component and it is possible that the changes invalidate existing client code. Hence, the component cannot be substituted with a different variant. This kind of component variability can be handled with a variable client implementation (e.g., using FOP as well). However, this increases complexity of client code and introduces dependencies between client configuration and component variant. To avoid this additional complexity, we argue that variability in the interface of an SPL has to be limited.

A special kind of interface variability occurs when a feature extends a method and thus changes the method's signature (e.g., by adding a parameter that is required for executing the method). In Berkeley DB, we observed that method signatures of a component may change due to reconfiguration, e.g., after extracting code of feature TRANSACTION into a separate feature module [RKSS07]. The feature introduces arguments into several methods. For example, there is a method `put` in Berkeley DB similar to the method we depict in Figure 4.9. In order to support transactions, a parameter of type `TXN` is required (Lines 2 and 5) for feature TRANSACTION. In the base implementation without any transaction code, the parameter is not needed and should not be part of the method signature (Line 2). This would cause variability of the signature of 94 methods between different configurations, i.e., with or without transactions.

Extensions of method signatures are not supported by FOP languages. The result is a fixed method signature that includes all arguments required in refining features, as shown in Figure 4.9. This is not an implementation issue but a general limitation

of the approach. If it would be possible for feature TRANSACTION to add argument `txn` to method `Put`, all existing method calls (calls in client code but also calls in SPL code) that do not expect the argument would still refer to a method without the argument. When we try to actually implement this variable signature, (1) we would have to overload the method or (2) we would have to replace the method with the extended method. Both possibilities do not solve the problem. When overloading the method, the existing method calls would refer to the unextended method. This would cause semantic errors because the extended method must be invoked. When replacing the method, the existing method calls are invalid because the unextended method does not exist in variants without the optional feature. Hence, we would also have to implement variability at the caller side (i.e., for every method call), which is generally possible with the C preprocessor (i.e., using `#ifdefs`).

In previous work, we analyzed different approaches to handle extensions of method signatures [RKSS07]. We found that the C preprocessor is also inappropriate for two reasons. First, it complicates SPL development (we have to provide variability at method invocations) and, second, the variable signature hinders interaction with other software that uses the SPL. Hence, we argue that such extensions must be avoided because they complicate SPL development and hamper use of SPL components. In our decomposition of Berkeley DB, we avoided variable method signatures by providing empty classes that are used as argument types. However, this means that we do not completely separate the features from each other. Furthermore, such extensions are hard to achieve if new features have to add arguments during SPL evolution. Variable argument lists as provided by C++ can also not be used to solve the problem because the order of the arguments matters and may change due to the feature composition order. In languages that support order independent argument lists, this could be a viable solution. For FeatureC++ and similar languages, we thus suggest changing the design of an SPL instead of using variable method signatures.

Avoiding variability of method signatures does not mean that a feature cannot overload an existing method. Overloading means that a new method is added and the existing method is still valid. For example, a feature TRANSACTION may overload a method `Put` by adding a method with an additional argument `txn`. In this case, however, the new feature has to ensure that the existing method `Put` still works. For example, the existing method may be refined to automatically create a new transaction on every method call. By contrast, extending the method means that the old method is not valid and cannot be used.

Component Polymorphism. The second problem caused by a variable component interface occurs when multiple components of an SPL must be used polymorphically. For example, when a client application uses different variants of a component, programmers face two major problems. First, they have to be able to distinguish component variants from each other and, second, they may want to write generic code that can be used with different variants. Both problems cannot be solved with FeatureC++ or the C preprocessor because the generated components are similar

but the interfaces are not subtypes of a more general interface. In Java, C++, and many other programming languages it is thus not possible to reference different component variants (generated from FOP code or with the C preprocessor) from the same source file. The reason is that generated component variants cannot be distinguished from each other. For example, a class `DB` of Berkeley DB has the same name in different variants. It is thus not possible for the programmer and the compiler to decide which of multiple variants to use.

Separating the component from its interface (as usual in CBSE) does not solve the problem because the component's interface is variable. That is, we are faced with the same problem of missing substitutability of component variants discussed above for component reconfiguration. We can solve the problem by providing subtype polymorphism of components or of their interfaces (i.e., subtyping between their interfaces). Before we describe possible solutions, we introduce the notion of *semantic* and *programming* interfaces for SPLs.

Semantic SPL Interface

We define the interface of a (specialized) SPL depending on the features that are included in all products [RSK10]. A specialized SPL is derived during staged configuration by deciding which features must be included in the variants of this specialization (i.e., selecting features; cf. Sec. 2.3.4).

We define the *semantic interface* of a (specialized) SPL as the set of features that are present in all valid instances of the SPL (i.e., the minimal set of included features). These are mandatory features, features selected via specialization, and features required due to constraints. The semantic interface thus defines the feature common to all products of the SPL. By adding features in specialization steps we extend the interface. For example, the semantic interface of the Berkeley DB SPL is the set of mandatory features. A specialization $BerkeleyDB_{TXN}$ that includes feature TRANSACTION has an extended semantic interface that includes this feature as well as features required due to constraints, such as feature LOGGING. However, not every specialization step extends an SPL's interface. For example, the interface does usually not change when we add a constraint that excludes a feature.

Subtyping. We define subtyping between SPLs at a conceptual level based on their semantic interface: When the semantic interface of an SPL *Derived* (i.e., the set of included features) is a superset of the interface of an SPL *Base* then *Derived* is a subtype of *Base*. Whether an SPL is a specialization of another SPL, and thus a subtype, can be checked with a SAT solver [TBK09]. Subtyping between SPLs allows us to use them polymorphically. For example, a client can use two different instances of Berkeley DB by accessing their common supertype. For a more detailed description of subtyping, we refer to [RSK10].

The expected and required semantic interfaces (i.e., the set of expected and required features) can be used to check whether a component provides the functionality required by another component. For example, we can check whether a concrete

Berkeley DB variant provides all the features required by an e-mail client that uses Berkeley DB. We only have to check whether the provided variant is a subtype of the required variant. This is a kind of semantic compatibility which is in contrast to the syntactic compatibility that is guaranteed by programming interfaces.

SPL Programming Interfaces

We define the *programming interface* of a (specialized) SPL according to its *semantic interface*. The programming interface is the union of the programming interfaces of the implementation classes that are defined in the features of the semantic interface. For example, the programming interface of Berkeley DB consists of the interfaces of all classes defined in mandatory features of the SPL. It does not include classes or methods introduced by optional features such as TRANSACTION because these features are not present in all instances. In specialization steps, we extend an SPL's programming interface up to a complete interface for a concrete component.

Subtyping. The semantic subtype relationship between specialized SPLs also applies to the interfaces of implementation classes [RSK10]. Hence, when SPL *Derived* is a specialization (and a subtype) of SPL *Base* then the interface $I_{Derived}$ of an implementation class $C_{Derived}$ defined in *Derived* is also a subtype of interface I_{Base} of the corresponding class C_{Base} defined in SPL *Base*. Consequently, a feature can modify the interface of a class only in a way that can be expressed with interface inheritance. For FOP this means that the refinement mechanism must correspond to subclassing. For example, a method refinement cannot change the signature of a method because this cannot be expressed with inheritance as we observed it when extracting feature TRANSACTION in Berkeley DB. Hence, an SPL should ideally follow our definition of the SPL programming interface to avoid problems with respect to client development but also with respect to SPL development. This could be part of an extended type system for SPLs. In combination with correspondence to the semantic SPL interface we provide a limited form of semantic substitutability [LW94]: A component variant can only be substituted with a different variant that has a compatible programming interface if it also provides a superset of the actually required features. That is, the new variant must also be a subtype of the required semantic interface. Hence, there can be components that have the same programming interface but are not in a subtype relationship with respect to their semantic interface. On the contrary, if a component is a semantic subtype of another component it is also a subtype with respect to the programming interface if the SPL satisfies our requirements on SPL interfaces.

4.3.2 Using Multiple Component Variants

When the implementation of an SPL adheres to our definition of semantic and programming interfaces, we can add features to a component variant without inval-

idating client code.[7] However, when using multiple variants, we have to provide a way to distinguish these variants from each other. For Berkeley DB, we observed this problem in client applications that use different variants.

Separating Component Variants. In C++, it is possible to use different variants of a component by referencing the variants in different source files of a client. This is possible because the files of the variants can be distinguished via their storage location using an `#include` directive. For example, it is possible to use two different variants of Berkeley DB by storing both variants in different folders and by including only one of the variants in each source file. This is not possible within a single source file. To overcome this limitation, we extended FeatureC++ to generate a distinct C++ namespace for each derived component variant [RSK10]. The namespace is used to access the classes of different component variants. For Java-based languages, such as Jak, the corresponding solution would be to use Java packages.

Avoiding Shared State. When using two different component variants, state should not be shared between the components. For example, the *Singleton* design pattern can be implemented with a static member variable [GHJV95]. The variable is used to store a single object of a class. When several SPL instances are used at the same time each instance requires its own object of the singleton class. Hence, the objects, and static member variables in general, must not be shared between different SPL instances to avoid inconsistencies. When two components are executed in separated processes this is automatically achieved. Using different components within the same client application, however, requires separating the state of the components. In FeatureC++, this is also achieved by generating namespaces.

Static Component Substitution. Using namespaces or packages to separate different component variants is sufficient when a client uses a single component that is reconfigured, i.e., statically substituted with a different variant. In this case, the old and the new component variants must have a common supertype that is *implicitly* defined via the methods the client accesses. This required programming interface should be made explicit by the client as it is usual in CBSE. However, this may still result in semantic incompatibility of the new component if it is not a subtype of the actually required semantic interface. Hence, the client should additionally define the required semantic interface (i.e., the minimal set of required features), which partially ensures semantic compatibility.

Support for Polymorphism and Subtyping. When using multiple component variants at the same time, it is a complex task to write generic client code that works with different variants. A possible solution is to use generics (e.g., C++ templates)

[7] This means that the client code will be syntactically correct and semantically correct with respect to the required features; but there may be semantic errors since we only ensure a limited form of semantic correctness.

in client development or to create an inheritance hierarchy of classes that wrap the different variants. Unfortunately, both solutions highly complicate client development. Another solution is to support polymorphism of the classes of different component variants according to the subtype relationship between the components as defined above. In this case, the different variants of the implementation classes of the component variants have to provide the subtype relationship of the components. In [RSK10], we proposed two solutions that address this problem using special code transformations to generate component hierarchies. The solutions are based on the concept of *family polymorphism* [Ern01] which is also supported by CaesarJ [AGMO06]. A detailed description of these approaches is outside the scope of this thesis. However, we want to note that we have to impose the restrictions on the implementation of a feature as described for subtyping with respect to the programming interface described above.

4.3.3 SPL Interface Design

We close this section with a proposal for designing SPL interfaces. If we aim at achieving polymorphism between component variants or if a client has to support reconfiguration of components, it is important to avoid interface variability that does not conform to interface inheritance such as variable method signatures. In general, variability of the interface of a class (adding a method) may also complicate client development. It is thus important to minimize the variability of the interface of an SPL. This can usually be achieved by implementing a feature's functionality in a set of classes with a fixed interface. Method refinements and delegation can then be used to connect a feature module (i.e., the set of classes) with a base program without changing the interface of the base program invasively (only classes are added). This also reduces the number of method extensions per method, which we often recognized to be a hindrance for readability and comprehensibility of FOP code. Hence, a feature ideally adds only classes to a base program and extends as few methods as possible. It should avoid to modify the interface of existing classes; if it modifies the interface, it should adhere to our definition of programming interfaces.

4.4 Related Work

Software development based on features was applied successfully in different domains [BO92, BCGS95, BJMvH02, GSC+03, XMEH04, LAS05, TBD06, LB06, ALS08]. However, there is less known about the impact on resource consumption and the applicability to embedded systems. We have analyzed resource consumption of static feature binding with FeatureC++ and could show that it can also be applied to develop SPLs for embedded systems. In the following, we present related approaches that aim at a similar goal using different implementation mechanisms.

Feature-oriented Programming. There are a few case studies on refactoring a program into an SPL with FOP. Trujillo et al. refactored the AHEAD Tool Suite into a

feature-oriented SPL and recognized a need for automation [TBD06]. They did not analyze resource consumption of FOP nor did they compare the variability mechanisms of FOP with the C preprocessor. During feature-refactoring Berkeley DB we also found that tool support is needed to provide a scalable approach. Kästner et al. proposed techniques to automate feature refactoring [KKB07]. Dreiling describes a semi-automatic approach for refactoring a program into an SPL using annotations [Dre10]. We semi-automatically translated C code annotated with the C preprocessor into FeatureC++ code but did not focus on automating feature refactoring (i.e., extraction of features), which was the second step in our case study.

Aspect-oriented Programming. There is a large amount of work on using AOP for developing customizable software systems. For example, AOP has been applied to DBMS [Ras03, NTN+04, TSH04], operating systems [CKFS01, CK03, LST+06], and middleware [ZJ03, ZJ04, CC04]. These studies show that AOP is also an appropriate technique to decompose software with respect to features. Evaluations of these solutions furthermore show that AOP can be used with negligible impact on performance and resource consumption, as long as no dynamic mechanisms are employed [GS05]. Our work is similar to these approaches and shows that FOP can be used to separate crosscutting concerns without impact on performance. Tešanović et al. examined AOP for DBMS customization [TSH04]. They evaluated their approach using Berkeley DB but have shown only customizability for small parts and not the whole system. Kästner et al. showed how the Java version of Berkeley DB can be refactored into an SPL with AOP [KAB07]. The approaches above did not compare variability mechanisms with the mechanisms provided by the C preprocessor. Some studies show that the comprehensibility of source code may be degraded when using aspects as it is suggested by studies that analyzed AOP [Ste06, KAB07, KK07]. We found that FOP may also degrade readability when compared to other variability mechanisms. Some studies propose that collaboration based designs like FOP should be preferred when developing software with respect to features [OZ05, LLO03, MO04, ALS06, AB06]. However, currently there is too less known about the comprehensibility of AOP and FOP code for a detailed evaluation.

Component-based Development. Development of customizable software has been in the focus of research since many years. Component-based approaches are also getting attention for DBMS development [GSD97, CW00, NTN+04, DG01b]. However, components have to deal with several difficulties: For example, when decreasing the size of components the communication overhead increases [CW00]. This can be avoided with a static configuration approach such as provided by FeatureC++. Furthermore, modularization of features into components is a complex task and sometimes not possible if the features are entangled with other features. Similarly, Griss et al. [Gri00] and Rashid [Ras03] argue that new techniques such as AOP should be combined with component technology to develop highly customizable software.

Components and AOP. In order to overcome the limitations of component approaches, component systems are combined with approaches that provide fine-grained customizability like AOP [KLM+97]. AOP allows for modularization of crosscutting functionality that is hard to encapsulate in a component. With COMET, Nyström et al. provide a component-based approach that uses AOP to tailor components by *weaving* customization code into a component if it is required [NTN+04]. The result is improved separation of concerns and fine-grained customizability without negative impact on performance and footprint. We introduced a definition for *variable SPL interfaces* that enable polymorphism of component hierarchies. Our definition of SPL interfaces can also be applied to a combination of components and AOP. In this case, an aspect would only be allowed to apply transformations that conform to subclassing.

Feature Annotation vs. Feature Composition. Kästner classifies SPL development approaches into annotative and compositional approaches [Käs10]. He compares annotation-based SPL development with compositional approaches and could show that disciplined annotations can also be used to achieve safe composition [KATS11]. However, also disciplined annotations require to either annotate all source files of a feature (which can be several files) or to use additional variability mechanisms such as provided by build systems. To achieve fine-grained extensions *and* modularity Kästner et al. propose to combine annotative and compositional approaches [KA08, Käs10]. This is in line with our findings when refactoring Berkeley DB.

Staged Configuration and Component Variability. Czarnecki et al. define staged configuration as a process that eliminates configuration choices [CHE05]. We additionally define a subtype relationship that corresponds to the product specialization hierarchy; it is the foundation for polymorphism and substitutability of component variants.

Component variability is handled by van Ommering using *diversity interfaces* [vO02]. A diversity interface defines not only required functions but also required properties needed for configuration of internal variability. Subtyping between interfaces is based on a subset relation of the elements of the respective interfaces [vO04]. This can also include diversity properties, which can be used to check whether the properties a component requires are provided by the system that contains the component. However, it does not define whether a component is a subtype of another component with respect to the features included. Variability of components was also expressed by van der Storm using the *component description language* (CDL) [vdS04]. CDL is based on the *feature description language* (FDL) [DK02] but additionally allows for describing dependencies between variable components using *requires* relations. Fries et al. present an approach to model compositions of multiple SPLs [FSSP07]. They use *feature configurations*, a selection of features, to describe a group of SPL instances that share these features. Feature configurations

are similar to SPL specializations but do not allow a user to describe multiple configuration steps. We use a specialization hierarchy to describe the commonalities of a set of specialized SPLs and reuse configuration decisions. All approaches above do neither define subtyping between component variants based on a feature selection nor a variable programming interface that depends on the feature selection.

Mixin Layers and Virtual Classes. Layered designs can be implemented with several techniques. Examples that support static binding are P++, a precursor of FOP [BDG+94] and virtual classes as supported by CaesarJ [AGMO06]. In contrast to AHEAD and FeatureC++, P++ explicitly defines layer interfaces. Static mixin composition with P++ and also mixin composition with virtual classes allow programmers to create different instances of a component at compile-time. However, both approaches require programmers to define a component in the host language. To derive different configurations of the same component, an additional composition mechanism has to be used (e.g., using the build system). CaesarJ can furthermore be used to achieve subtyping between different component variants of an SPL. However, as we analyzed in previous work [RSK10], the mixin composition process imposes a feature composition order, which makes it sometimes impossible to achieve the desired type hierarchy and a correct composition order at the same time.

Nested Intersection. The language J& supports composition of multiple components using nested intersection [NQM06]. It is based on composition of classes and packages with their inner classes similar to virtual classes. J& may be better suited for implementing component hierarchies than virtual classes because it defines *static* virtual types, which are attributes of packages or classes and not of objects. However, the composition mechanism of J& does not linearize class extensions, which complicates development of independently composable features.

4.5 Summary

In this chapter, we presented optimizations for FeatureC++ that allow us to statically bind features without any impact on resource consumption. In a case study, we have shown that FOP can replace the mechanisms to implement variability in Berkeley DB. These are conditional compilation, build scripts (GNU Make), macros, and function pointers. However, we have also shown that macros provide implementation mechanisms, not related to variability, that we cannot easily replace by OOP or FOP mechanisms. Furthermore, function pointers are used to implement static and dynamic variability which can be replaced by FeatureC++ refinements. Until now, we only considered static binding but we introduce dynamic binding of feature modules in the next chapter. Qualities of the source code like *separation of concerns* and comprehensibility are important with respect to maintainability and extensibility of software. We cannot easily compare annotations and FOP without extensive experiments [FKAL09], but we could show that we can replace multiple

variability mechanisms with a single approach, namely feature modules. Compared to the C preprocessor, FOP improves reuse possibilities because a feature module can be combined with other independently developed feature modules that provide the same interface. Furthermore, with FOP we can avoid syntactic and some kinds of semantic errors by using an SPL-aware type system [AKGL10]. It is important to note that type safety can also be achieved with an annotative approach and appropriate tool support [Käs10], but this is not supported by the C preprocessor. We have also shown that, similar to the C preprocessor, static composition of feature modules avoids any negative impact on performance or footprint. Our analysis shows that static binding may cause a high functional overhead with respect to binary size. With respect to performance, we observed only a small overhead because most features of Berkeley DB can be configured at runtime are thus only used when needed.

Our comparison does not mean that FOP is better suited for developing SPLs. For example, it may be better for a company to stay with a preprocessor-based approach to avoid the effort for a migration to FOP (e.g., to FeatureC++) [KA08]. Hence, it has to be analyzed for each concrete scenario which approach should be applied. Our comparison provides a basic guideline for such decisions. Furthermore, we argue that a combination of FeatureC++ and the C preprocessor can be useful. Especially some uses of macros and annotation of single statements that are not reused may be beneficial to simplify SPL development.

We have shown that current FOP approaches and also other SPL implementation techniques lack appropriate support for implementation of component SPLs. Variable method signatures and, more generally, a variable interface complicate development and use of component SPLs. We thus introduced a variable SPL interface consisting of a semantic and a programming interface. We argue that an SPL should adhere to the extensibility limitations imposed by our interface definition to minimize interface variability and thus reduce the complexity of developing and using SPLs. Consequently, tools for SPL implementation (i.e., preprocessors, compilers) have to check whether a component SPL satisfies requirements of a variable interface.

In Berkeley DB, we applied static composition on top of dynamic binding of some features. Thus we achieve both, flexibility due to dynamic binding and improved performance when dynamic binding is not needed. However, this approach does not integrate static and dynamic binding very well and increases the development effort: Programmers have to encapsulate a feature in a feature module and, at the same time, they have to implement dynamic binding, e.g., using conditional statements. In the next chapter, we present an extension of FeatureC++ to support also dynamic binding for feature modules.

5 Dynamic Binding of Feature Modules

This chapter shares material with the GPCE'08 paper "Code Generation to Support Static and Dynamic Composition of Software Product Lines" [RSAS08].

FOP provides means to modularize the features of an SPL already at the language level. It provides a mechanism for class extensions which is independent of the binding time. This allows us to bind features not only in a preprocessing step but also at runtime. In contrast to static binding of feature modules, dynamic binding requires to compose the classes of a program according to a feature selection at runtime. This composition process can be realized by compiling the variant at runtime and applying the changes to the running program [PKG+09]. We avoid compilation at runtime by generating *binary* feature modules that are dynamically composed with each other similar to components. This has several benefits. First, composition is much faster because the feature modules do not have to be compiled; the binary modules are loaded on demand (or already at program startup) and are connected with other feature modules. Second, composition of binary feature modules allows us to reconfigure an SPL at runtime. For example, we can load additional feature modules without regenerating the whole SPL. In a first step, we focus on composition of features *before* executing the SPL code. The binding may occur during runtime of a program (e.g., a client program that uses a DBMS SPL) but it does not change after execution of the SPL code has started. Hence, it is binding at load-time with respect to the SPL [KCH+90]. To support reconfiguration of an already running SPL instance, existing classes have to be extended with new code when a feature is activated and existing objects have to be extended with new data. Hence, it is important to distinguish between binding at SPL load-time and at runtime (i.e., runtime adaptation).

In this and the next chapter, we focus on dynamic composition. However, the presented code transformations can also be applied for binding at runtime, which we discuss in Chapter 7. Based on the concept of feature modules we now analyze requirements on a language that supports both, static and dynamic binding. We then present code transformations to generate binary feature modules that can be bound dynamically.

5.1 Language Support for Dynamic Feature Binding

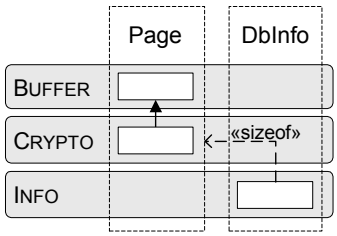
While static binding is well supported by Jak and FeatureC++, there are language constructs that cannot be used when dynamic binding is applied. An example is the C++ compiler construct `sizeof`, which calculates the size of an object at compile-time. This is not possible when the size of an object changes at runtime due to a reconfiguration. Such language features also exist in other programming languages such as Java (e.g., enumerations). In the following, we we analyze how C++ constructs can also be supported when dynamically binding feature modules of FeatureC++.

5.1.1 Compile-Time Constructs

Compiler constructs are frequently used because they can be evaluated by the compiler and the compiler can optimize the program code according to the evaluation result. Hence, such constructs should also be available in a language that supports both, static and dynamic feature binding. A solution is to support the constructs only when they are applied to statically composed language elements. In general, such constructs can also be evaluated at runtime (e.g., calculating the current size of an object), but the result of the evaluation is usually already required at compile-time. Furthermore, the result of the evaluation (e.g., the size of an object) may change when additional features are bound at runtime.

Calculating the Object Size with `sizeof`

In C++, the size of an object can be determined with `sizeof` at compile-time. In FeatureC++, the `sizeof` construct can only be applied to statically composed classes and the programmer gets a compiler error otherwise. This results in implementation constraints that limit the variability of an SPL. We depict an example in Figure 5.1 for a class `Page` of a DBMS which has a variable size. The class is extended in feature Crypto, which adds variable `bIsEncrypted`. Class `DbInfo` in feature Info uses `sizeof` to calculate the size of the class. Using static binding, the size depends on the feature selection but can be computed by the compiler because configured features are already known at compile-time. Using dynamic binding, the size of class `Page` may change at runtime and `sizeof` is not applicable. Hence, when feature Info is used, feature Crypto cannot be configured dynamically. That is, when binding all features dynamically, features Info and Crypto are mutually exclusive. This is problematic when both features are required and can only be avoided with a different implementation. We do not support evaluation of `sizeof` at runtime because the programmer would expect that the computed size will be correct and does not assume that it may change.



Feature BUFFER

```
1  class Page { ... };
```

Feature CRYPTO

```
2  refinex class Page {
3    bool bIsEncrypted;
4  };
```

Feature INFO

```
5  class DbInfo {
6    void ShowInfo(Page* p) {
7      size_t size = sizeof(Page);
8      ...
9    }
10 };
```

Figure 5.1: Using the `sizeof` operator with composed classes. The size of an object of class `Page` depends on the feature selection.

Type Declarations

In C++, a type alias can be declared with keyword `typedef`. In FeatureC++ an alias can also be declared within a feature. Hence, the alias depends on the feature selection. It may be undefined if the feature is not present or it may be defined differently in different features. When using the alias in another feature the type has to be known at compile-time (e.g., for static type checking). For example, to support different string encodings, the character type of a string could be defined as an 8 bit value in one feature (e.g., to support UTF-8 encoding) and as a 16 bit value in another feature (e.g., to support UTF-16 encoding). Such varying definitions can be easily handled with static binding. With dynamic binding, however, a variable `typedef` causes an error when used by another feature because it depends on the SPL's configuration which is not known at compile-time. Hence, typedefs that cause a binding error (1) must be defined outside of dynamically bound features as separate types, (2) must be used local to a feature, or (3) must be avoided (e.g., using a class to wrap all possible types).

Templates

C++ templates are statically instantiated at compile-time. When using dynamic binding in FeatureC++, a template may be defined with a dynamic type as an argument. For example, `template <class Dynamic>` is a valid template definition even though class `Dynamic` is composed dynamically. In FeatureC++, the template specification of a template class must be declared in the base class, not in a class refinement. Consequently, the template arguments of a class do not change depending on the feature selection. This allows us to support dynamic binding for template classes. For example, it is possible to compose a class `template <typename T> class Dynamic {...}` dynamically. In our prototype implementation we do not support dynamic binding for template arguments or tem-

plate classes.

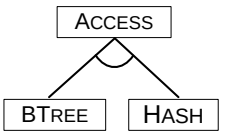
Enumerations

In C++, enumerations define constants that support static type checking [Str94]. An enumeration is implemented as an integer and each defined constant of an enumeration corresponds to a value of the integer at runtime. Static type checking ensures that only valid constants are assigned to a variable of an enumeration type. In FeatureC++, an enumeration can be extended in the same way a class is extended. New features can add constants to an enumeration which can be checked by the compiler if using static binding. As long as there are no conflicting constant definitions between features (i.e., two mutually exclusive features define the same constant with different values), extension of enumerations can also be supported for dynamically bound SPLs. In this case, enumerations in all features that are selected for dynamic binding are composed statically independent of the dynamic feature selection. This results in a single enumeration with distinct constants for all dynamically bound features. When using an enumeration that is extended in dynamically composed features, the compiler can statically check if a given enumeration value exists (i.e., if it is defined in one of the dynamically bound features). However, the compiler cannot decide whether an expression with an enumeration value is valid with respect to the dynamically bound features. That is, a given enumeration value is invalid at runtime if the corresponding feature is not available at runtime. This could be statically checked for SPL code using an SPL-aware type system. For client code that uses an enumeration type of a dynamically bound SPL, this can be checked at runtime.

5.1.2 Static Class Members

As already discussed for static binding, static data members of a class should not be shared between different SPL instances (cf. 4.3.2). This also applies to dynamic binding and allows programmers to create multiple SPL instances (e.g., different instances of a component) in the same program at runtime. Hence, a static member that is defined in a feature should not be shared between all instances that use this feature. Static variables thus have to be treated as members of an SPL instance to avoid interaction with other instances.

Furthermore, the implementation of static methods can vary between different SPL instances. For example, in FeatureC++ we can refine a static method, as shown in Figure 5.2. Method `CreateIndex` returns an index data structure that corresponds to the feature selection of the SPL. It returns a B-Tree index when feature BTree is active and a hash index when feature Hash is active. The result of a call to `Index::CreateIndex()` thus depends on the currently active feature selection. With static composition this is automatically achieved because only one of the implementations is available in a concrete product and products are separated from each other (e.g., via namespaces). With dynamic composition, however,



Feature BTREE

```
1  refines class Index {
2    static Index* CreateIndex() {
3      return new BTreeIndex();
4    }
5  };
```

Feature HASH

```
6  refines class Index {
7    static Index* CreateIndex() {
8      return new HashIndex();
9    }
10 };
```

Figure 5.2: Refining static methods in FeatureC++.

a static method has to be static with respect to an SPL instance, which is defined at runtime. The implementation of the method can be different for differently configured instances and may even change when an SPL is reconfigured at runtime. Hence, the correct implementation has to be chosen at runtime similar to virtual methods. Since the type of the class is variable (it depends on the enclosing SPL instance) virtual types can be used to solve this problem as described in the previous Chapter (cf. Sec. 4.3.2). We use an emulation of virtual types, as we describe in Section 5.3.1.

5.1.3 Object Allocation on the Stack

There are different ways to create an object in C++. Additionally to using the `new` operator, an object can be created on the stack. Allocation on the stack simplifies instantiation of local objects because the object is destroyed after leaving the scope of the block it was allocated in. For example, an object allocated on the stack is automatically released when an exception is thrown without the need to delete it manually. This not only avoids memory leaks but can also be used to allocate other resources when it is important to release them after use (e.g., locking concurrently accessed data). Furthermore, allocation on the stack usually performs much better than allocation on the heap because no memory has to be reserved.[1]

To allocate the memory of an object on the stack the compiler needs to know the object size. This is not possible in case of dynamic binding because the size cannot be determined before load-time. To avoid different semantics between static and dynamic binding, object allocation on the stack has to be emulated when using dynamic binding. We achieve this by using a proxy [GHJV95] that is allocated on the stack. The proxy refers to the actual dynamically composed object and deallocates the object when leaving the scope.

[1] When allocating memory on the stack, only the stack pointer has to be changed, which usually does not result in a memory allocation via the execution environment.

Feature CORE

```
1  class DB {
2    bool Put(Key& key, Value& val) { ... }
3  };
```

Feature QUERYENGINE

```
5  refines class DB {
6    QueryProcessor queryProc;
7    bool ProcessQuery(String& query) {
8      return queryProc.Execute(String& query);
9    }
10 };
```

Feature TRANSACTION

```
12 refines class DB {
13   Txn* BeginTransaction() { ... }
14   bool Put(Key& key, Value& val) {
15     ... //transaction specific code
16     return super::Put(key,val);
17   }
18 };
```

Figure 5.3: FeatureC++ code of class `DB` decomposed along the three features CORE, QUERYENGINE, and TRANSACTION.

5.1.4 Summary: Limitations of Dynamic Binding

We have shown that dynamic binding cannot directly be applied to all C++ language constructs that are intended for static composition. Similar limitations exist for other languages. We presented solutions for some constructs (e.g., object allocation on the stack), but we have to forbid application of some constructs in case of dynamic binding (e.g., using `sizeof`). In this case, programmers have to change the design of an SPL to avoid such low-level constructs or they may use only static binding for such features, as we present in the next Chapter.

5.2 Dynamic Binding of Feature Modules

In order to support dynamic binding of features implemented with FOP, the classes of an application have to be composed according to the active features at runtime. As an example, reconsider class `DB` (cf. Fig. 5.3) of a DBMS SPL. The base class has to be extended dynamically with code of feature TRANSACTION when activating the transaction management. To support this, we generate a *binary feature module* for each feature. The module contains all classes and class refinements of the feature and can be dynamically composed with other binary feature modules. In the following, we present the required code transformations, which are based on message forwarding between the feature modules.

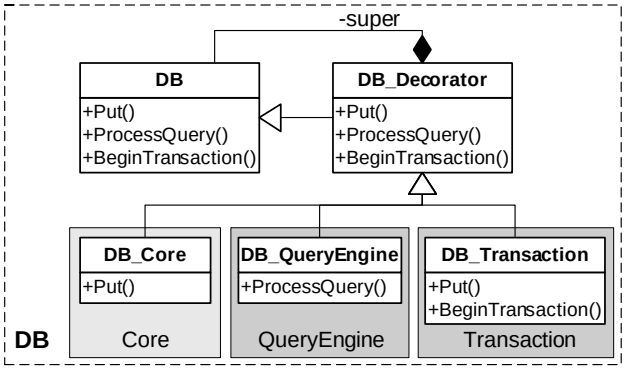


Figure 5.4: Class diagram of the generated decorator hierarchy for dynamic binding of class `DB` using features QueryEngine and Transaction.

5.2.1 Dynamic Binding with Decorators

We support dynamic binding of feature modules in FeatureC++ using a code transformation that transforms the feature modules into dynamically composable fragments. Hence, the classes that are crosscut by several feature modules have to be composed at runtime. We thus transform the refinement chain of a class into a chain of objects that are connected via the *decorator* design pattern [GHJV95]. At the same time, each class fragment is part of a feature module. Hence, a class fragment corresponds to a role in role-based approaches and a feature module corresponds to a collaboration that contains roles of different classes. In the following, we first describe the transformation required to generate dynamically composable class fragments. We then describe how we enable dynamic composition of entire feature modules.

Dynamic Composition of Classes

Similar to the *Delegation Layers* approach [Ost02], we use the decorator pattern to compose classes dynamically. Each dynamically composable class (i.e., classes that are crosscut by multiple feature modules) consists of a decorator for each refinement and an object is composed dynamically by combining instances of the decorators.

For illustration, we depict the class diagram of the transformed class `DB` in Figure 5.4. The class is composed from its refinements, which have been transformed into decorators (`DB_Core`, `DB_QueryEngine`, `DB_Transaction`), each belonging to a separate feature. The generated decorator interface (class `DB`) is used to reference dynamically composed classes within the transformed code and also from external client code. The abstract decorator class `DB_Decorator` maintains a reference to the predecessor refinement (`super` reference) and forwards operations that are not implemented by a concrete decorator. The implementation of methods and their refinements are located in the concrete decorators. For example, method `Put` (Line 2 in Figure 5.3) and its refinement in feature Transaction (Line 14) are transformed

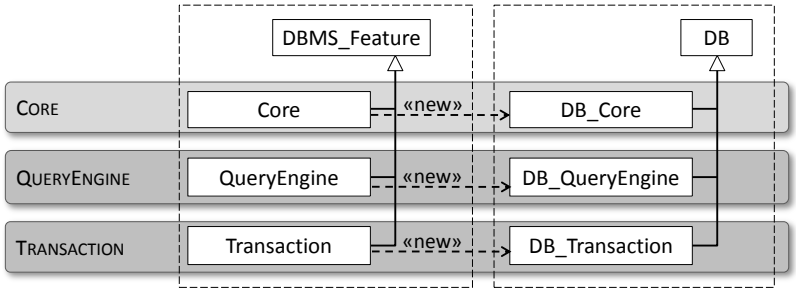


Figure 5.5: Implementing features (left side) and an implementation class (right side) in dynamically composable SPLs as a chain of decorators.

into methods of concrete decorators `DB_Core` and `DB_Transaction` respectively (cf. Fig. 5.4). Method refinements invoke refined methods by using the `super` reference of the decorator class.

Feature Classes

When dynamically creating an SPL instance, we instantiate and compose a set of features at runtime. We support feature instantiation and composition at the code level by representing features as classes. In the following, we call the classes that represent features *feature classes* to distinguish them from *implementation classes*, i.e., the classes that implement an SPL. A feature class is only responsible for instantiation and composition of the refinements of the corresponding feature.

Feature classes correspond to the enclosing classes in the Delegation Layers approach but are generated by the FeatureC++ compiler. Much like dynamic composition of implementation classes, we combine multiple feature classes using the decorator pattern, as shown in Figure 5.5. For each feature module (CORE, QUERYENGINE, and TRANSACTION) we thus generate a feature decorator. All decorators inherit from an abstract decorator, which represents an arbitrary feature of the product line (`DBMS_Feature` in Fig. 5.5). Each feature instance, i.e., an instance of a feature class, maintains a `super` reference to the predecessor feature in a composed program. All feature modules are compiled separately. The resulting binary modules are merged into the binary of the program or are deployed as separate binaries, one for each feature module (e.g., as Windows DLLs).

Static Preselection of Features

We reduce the number of features for dynamic composition by statically preselecting only required features. That is, a user selects the features that have to be included in the generated dynamic SPL. By selecting only required features, we reduce the binary code size, the size of objects, and the number of methods in the dynamic interface of a class. This simplifies client development because only the actually needed classes and methods are presented to the client developer. Furthermore,

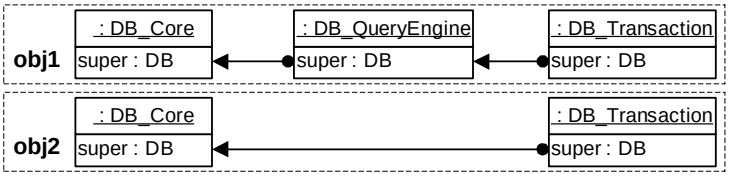


Figure 5.6: Object diagrams of instances of class `DB` for two different feature selections.

static preselection is required to exclude features that cannot be compiled due to platform limitations. For example, we have to decide which operating system and compiler should be used before compilation.

Feature Interactions

When two features interact, it may be needed to modularize the interaction code in a *derivative* as described in Section 2.2.5. Using static binding, all required interaction code is statically composed into a single program. This is more complicated when using dynamic binding. In this case, interaction code is also bound dynamically. Hence, we generate a single binary interaction module for each derivative, which is only bound when the corresponding features are bound. For example, transaction specific code of the query engine of our DBMS example in Figure 5.5 is only needed when features QueryEngine and Transaction are used at the same time. Hence, we generate a corresponding feature interaction module that is loaded only when both features are loaded. The generated module is not different from a regular feature module.

5.2.2 Using Dynamically Composed Classes

Class Instantiation. Instantiation of dynamically composed implementation classes means to create objects of the generated concrete decorator classes according to the selected features. In Figure 5.6, we show two different instances of class DB using the Core implementation as well as features QueryEngine and Transaction. Each instantiated refinement contains a `super` reference that points to the next refinement in the chain. A dynamically composed object can be used in the same way as an instance of a regular class and can be modified at runtime by adding or removing instances of decorators. The refinement chain thus corresponds to a linked list of object fragments (i.e., roles). Changing the configuration of an object corresponds to insertion, exchange, and deletion of elements of this *refinement list*.

For class instantiation, a feature decorator provides a factory method for each class refinement of the feature. Each factory method is responsible for creating objects of the corresponding decorator (cf. `new` in Figure 5.5). It combines created decorator instances with objects created by parent features (i.e., the features above

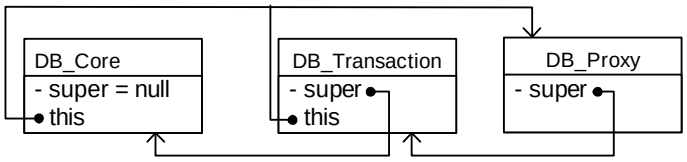


Figure 5.7: Layout of class `DB` including a proxy that forwards data to the first refinement.

it in the stack of features of an SPL instance). For example, to create object `obj1` of Figure 5.6, feature class `Core` creates an instance of `DB_Core`; `QueryEngine` combines the new object with an object of `DB_QueryEngine`. Feature class `Transaction` combines the compound object with an instance of `DB_Transaction` resulting in the final object. If there is no instance of feature class `QueryEngine`, then no instance of `DB_QueryEngine` is created, which results in `obj2` (cf. Fig. 5.6).

Method Invocations and the Self-problem. Each object is wrapped by a proxy that forwards method calls to the outermost refinement. As an example consider `obj2` from Figure 5.6 including its proxy as shown in Figure 5.7. Calls to method `Put` invoke the proxy which forwards operations to decorator `DB_Transaction`. We use the proxy for object allocation on the stack (cf. Sec. 5.1.3) and for binding at runtime. In case of binding at runtime, the proxy is required because the object may be extended with additional decorators. For that reason, each decorator stores the *object's* `this` pointer that refers to the proxy instead of itself. This solves the *self-problem*, which occurs when an object is spread over several smaller parts [Lie86]: When a dynamically composed object invokes a non-private method of itself it invokes the method of the proxy object because subsequent decorators may override the method. Private methods cannot be overridden in refinements and are always implemented in the decorator itself. If runtime adaptation is not required, we omit the proxy to reduce the object size. For allocation on the stack and for method invocations we then use the first decorator of an object.

5.3 Feature Instantiation and Composition

An SPL instance is composed from code at a meta-level that must be statically bound before SPL instantiation. For example, a client application may create an instance of a DBMS SPL at runtime. As illustrated in Figure 5.8, we distinguish between two scenarios: (a) a component, a library, or a framework is developed as a *component SPL* and is composed and initialized from an external client application and (b) an *SPL application* (i.e., a stand-alone program) manages feature composition and initialization by itself. We describe both scenarios in the following.

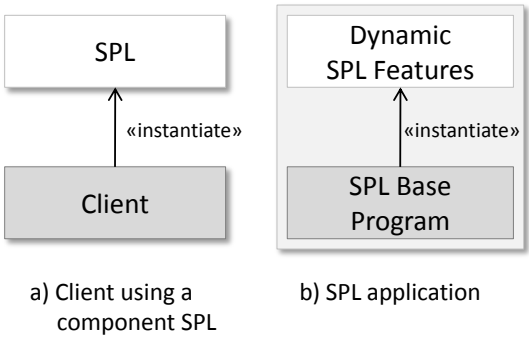


Figure 5.8: Instantiation of an SPL developed as a component (a) and as a stand-alone application (b). Instantiation code at the meta-level is shown in gray; instantiation is depicted with an arrow.

Component SPLs. In Chapter 4, we analyzed how clients should use a component that is developed as an SPL. Due to dynamic binding of the component with FeatureC++, a client can create the required instances at runtime (cf. Fig. 5.8 a). For this purpose, the client itself can be implemented with FeatureC++ or with plain C++. This is possible because FeatureC++ generates C++ code that a can be directly accessed from C++ code. A client can create instances of the feature classes manually or may use semi-automatic feature composition supported by generated code. Using the semi-automatic approach, the client calls a generated SPL method that returns an SPL instance according to a list of selected features. In the next section, we describe manual and automatic composition in detail.

SPL Applications. When a stand-alone application is developed as an SPL there has to be initialization code that can be executed at application startup. In C++ or Java, a global `main` function is invoked at program startup and acts as the entry point for program execution. In FeatureC++, this `main` function can be defined in any feature outside of a class and is automatically invoked by the runtime environment when using static composition. Using dynamic feature binding, one of the features (usually the base program) has to provide this method such that it can act as the application's entry point (cf. Fig. 5.8 b). Similar to a client application that uses an SPL component, the main method can be used by the programmer to compose the application features manually or the `main` method is generated by FeatureC++ to support automated composition. Hence, an SPL application corresponds to a component SPL that is embedded into a generated client application which provides initialization code only.

Safe Composition. Independent of the used SPL type (component or stand-alone application), we have to ensure safe composition at runtime. We achieve composition safety by composing the feature modules according to the variability restrictions

```
1  bool Store(DBMS& dbms, DB& db, string strKey, string strVal) {
2    Key* key = new (dbms) Key(strKey);
3    Value* val = new (dbms) Value(strVal);
4    return db.Put(key,val);
5  }
```

```
6  int main() {
7    //create differently configured DBMS
8    DBMS dbmsPersist = Persistent(Core());
9    DBMS dbmsInMem = InMem(Core());
10
11   //create different databases
12   DB* persStorage = new(dbmsPersist) DB();
13   DB* inmemStorage = new(dbmsInMem) DB();
14
15   Store(persStorage,"key","value");
16   Store(inmemStorage,"key","value");
17 }
```

Figure 5.9: Source code of a client application using dynamic composition of different DBMS and emulation of virtual classes.

defined in the feature model. We describe details about the safe composition mechanisms in Section 5.3.3. The mechanism is used for manual and for automated feature composition.

5.3.1 Manual Feature Composition

Manual feature composition means to create instances of feature classes and to compose the created objects (i.e., the feature instances). In Figure 5.9, we show an example for dynamic composition of the DBMS SPL in a client application. When the SPL is a stand-alone application, the same code can be implemented as part of the SPL itself. In this case, the code has to be defined in the base program.

In the `main` method in Figure 5.9, we create a DBMS for persistent storage by instantiating feature `Core` of the SPL (Line 8) and feature `Persistent` that implements persistent data storage. An in-memory variant is composed from features `Core` and `InMem` (Line 9). Implementation classes of the SPL must be instantiated by providing an SPL instance object and cause a compile-time error otherwise. This is done by using the overloaded `new` operator, which receives an instance of the SPL that is to be used for object creation (Lines 12, 13). The new operator invokes the factory method of the corresponding SPL instance. Since the type of a class depends on an SPL instance, this is an emulation of virtual classes with C++. In our example, the type of class DB depends on the type of an instance of the DBMS SPL. However, since there is no representation of virtual types in C++, we use interface `DB` to refer to all variants of the class.

```
1 bool Store(DB& db, string strKey, string strVal) {
2   Key* key = new (db.GetInstance()) Key(strKey);
3   Value* val = new (db.GetInstance()) Value(strVal);
4   return db.Put(key,val);
5 }
```

Figure 5.10: Creating objects that correspond to the SPL instance of another object.

Polymorphism of SPL Instances. In client code, the feature classes, and thus different components, can be used polymorphically as shown for argument `dbms` in method `Store` (Figure 5.9, Line 1). The `DBMS` interface represents any possible SPL instance that can be dynamically created. SPL classes can be instantiated via objects of the abstract SPL type without specifying the concrete type of a DBMS instance. This is shown for classes `Key` and `Value` in method `Store` (Lines 2–3). The created objects `key` and `val` correspond to the dynamic type of `dbms`. Thus, storing data in Lines 15 and 16 results in different instances of classes `Key` and `Value` for a persistent and an in-memory DBMS.

In contrast to virtual type support by a compiler, this solution provides reduced static type-safety because the implementation is based on C++. For example, we cannot completely type check the call to method `Put` in Line 4. The compiler ensures that the arguments `key` and `val` have the correct type with respect to classes `Key` and `Value` but it does not ensure that these are the correct *variants* of the class with respect to the receiver of the method (object `db`). In this example, we cannot ensure that argument `db` is also an object of SPL instance `dbms`. This is a problem if object `db` expects methods that are not defined in the concrete variants of classes `Key` or `Value`. Invocation of method `Put` thus causes a runtime error if objects `key` or `value` are not objects of the same SPL instance as `db`. In Figure 5.10, we show a different implementation of method `Store` that avoids this problem. We use method `GetInstance()` of object `db`, which returns the SPL instance of an object. This ensures that the created objects `key` and `val` are compatible with `db`. Nevertheless, the call to method `put` stays unchecked by the compiler. This can only be avoided with a programming language that supports virtual types such as Object Teams [Her02].

Note that type-safety is only a problem in client code that uses multiple instances of the same SPL. In clients that use a single instance only the problem does not occur because there is only one variant of each class. The problem also does not occur inside an SPL (i.e., in method calls within the SPL code) even when using multiple instances. Method calls within the SPL are type safe because all method invocations refer to the same instance. This is ensured by FeatureC++ when generating a dynamic SPL.

The provided polymorphism of feature classes does not completely satisfy the requirements we defined in the previous chapter to handle component variability using polymorphism of component variants. In contrast to the solution we proposed in Chapter 4, there is no subtyping between components according to a hierarchy

but only between a general SPL interface (which is a union of the interfaces of all features) and the concrete components. Support for a complete hierarchy requires support for virtual classes.

Since the generated SPL interface is a union of the feature interfaces (the public and protected methods of each class are merged into a single interface), a class refinement cannot modify the signature of an existing method as required by our interface definition. However, this general interface hinders some SPL implementation patterns that would be valid when using virtual classes. The reason is that all variants of a dynamically composed class have to conform to the same interface. That is, also mutually exclusive features cannot introduce incompatible methods. Two method signatures are incompatible when they cause a type error, i.e., when the method cannot be overloaded due to *incompatible* arguments or return types.[2] This limitation may force developers to change the SPL design to avoid this kind of variability (e.g., using a separate class to encapsulate the variable element). To solve this problem, FeatureC++ has to generate code in a language that supports virtual classes instead of plain C++ code. Consequently, the client must also be implemented in a language that supports virtual classes, such as an extension of FeatureC++, as we proposed in [RSK10].

Our approach for safe composition can be combined with static type checking of the entire SPL [AKGL10], which allows us to ensure type-safe dynamic composition. Compared to a type system for virtual classes (e.g., as in CaesarJ), static type checking of an SPL ensures type-safety for the entire SPL and not for created SPL instances.

5.3.2 Automated Feature Composition

Static composition of SPLs is usually provided by tools that allow programmers to select features and validate a feature selection based on the description of an SPL [Bat05, pur04, Kru08]. This is usually not the case if dynamic composition is used, even if a feature model is available. Hence, the developer of a client application is responsible for validating the consistency of an SPL instance, which is tedious and error-prone.

To ease SPL instantiation we developed *FeatureAce* (Feature Adaptation and Composition framEwork), a customizable framework that supports composition of features at runtime. FeatureAce itself is developed as an SPL and can be tailored to the requirements of the application scenario. Besides basic mechanisms to compose features it supports adaptation of SPLs at runtime and provides advanced capabilities for self-configuration as we present in Chapter 7.

When composing feature modules at runtime, the configuration process has to consider the constraints defined in an SPL's feature model. We thus integrate the feature model in the form of meta-data into the running SPL to validate configurations before applying them. *FeatureAce* provides functionality to automate creation

[2] This applies only to protected and public methods that are part of the SPL's interface.

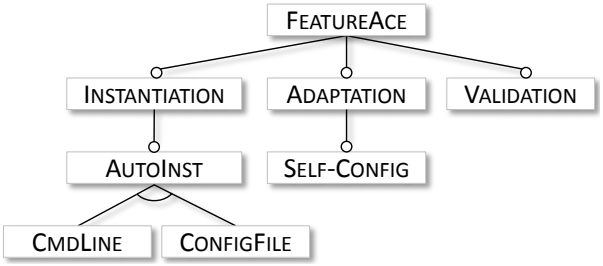


Figure 5.11: The feature model of FeatureAce. Customization of dynamic product instantiation and runtime adaptation capabilities is achieved by selecting the required features.

of SPL instances within a running program, to validate a feature selection according to the feature model before instantiation, and to modify a running SPL instance, which is subject of Chapter 7.

Dynamic Composition with FeatureAce

FeatureAce supports automated instantiation of SPLs using a core implementation and an SPL-specific code generation process at compile-time. The generated code is similar to the instantiation code presented in Figure 5.9. FeatureAce creates SPL instances by composing instances of feature classes according to a configuration provided in the form of a list of features. It validates a feature selection with respect to the feature model, derives the correct feature composition order, selects corresponding feature modules, and creates an SPL instance by composing the feature modules. To support a set of different instantiation scenarios, we have developed FeatureAce as a product line itself. This allows us to generate the appropriate code for composition and validation tailored to the needs of an SPL or an application scenario. For example, FeatureAce supports different ways for dynamic instantiation implemented as distinct features. This flexible composition mechanism allows us to generate a tailor-made dynamic composition infrastructure for the SPL by selecting the desired features of FeatureAce.

In Figure 5.11, we show the feature diagram of FeatureAce. Feature AutoInst encapsulates the functionality required for automated SPL instantiation using command line arguments (feature CommandLine) or a configuration file (feature ConfigFile) to provide an initial feature selection. Features Adaptation and Self-Config provide functionality for runtime adaptation, as we describe later. Feature Validation provides functionality to check validity of a configuration at runtime before composing the feature modules. The feature modules of FeatureAce consist of code implemented with FeatureC++ and transformation rules that modify the FeatureC++ code transformation process.

We connect SPL implementation and the compositional meta-level of FeatureAce by providing metaprogramming support for feature modules. This means that a

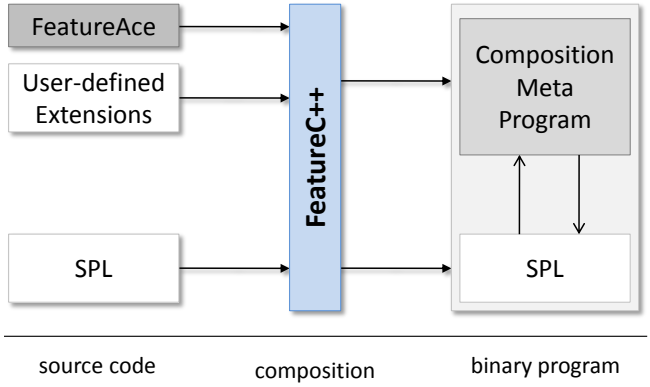


Figure 5.12: Generating a dynamic SPL from FeatureAce (a framework for feature-based composition and adaptation), user-defined extensions of FeatureAce, and an SPL.

feature can be accessed at runtime using a reflection API (e.g., for retrieving information about the feature or modifying an active SPL instance). In Figure 5.12, we depict the code generation process for deriving a runtime adaptable SPL. It is generated by the FeatureC++ compiler from an SPL's implementation and FeatureAce. In the resulting binary SPL, a *metaprogram* is responsible for composition at runtime. Its source code is part of FeatureAce and it is generated as a subsystem of the whole program. The metaprogram is bound to the SPL via generated SPL-specific code. To support arbitrary scenarios (e.g., loading a configuration from network), a programmer can create user-defined extensions of FeatureAce (shown on the left side of Figure 5.12). The extensions are implemented as additional feature modules without the need for invasive modifications of FeatureAce.

Reconfiguration at Runtime

So far, we only addressed composition of features at load-time (i.e., before executing their code). Dynamic binding can also mean to change the configuration of an already instantiated SPL. This is also known as runtime adaptation and is subject of dynamic SPL research [HHPS08a]. We can apply the presented approach also to runtime adaptation because the required operations on the level of features and class instances are addition, removal, and exchange of decorators. At the level of features it means to exchange feature instances (implemented as decorators); at the level of implementation classes it means to modify existing objects (instances of implementation classes) that are composed from a set of decorator instances. Both, feature instances and instances of implementation classes, are thus linked lists (cf. Fig. 5.6) that can be easily modified. In Chapter 7, we describe an extension of FeatureAce with support for adding, removing, and exchanging features of running SPL instances.

5.3.3 Composition Safety

Verifying a composition of features at runtime is not different from static verification. Based on an SPL's feature model, FeatureAce validates program configurations at runtime. It detects violations of constraints such as invalid feature combinations and thus achieves composition safety with respect to the model constraints. As the feature model and additional boolean constraints (e.g., FeatureX *requires* FeatureY) can be transformed into a propositional formula [Bat05], we use a SAT[3] solver to test whether a valid variant can be derived from a feature selection or not. Even though the SAT problem is NP-complete, satisfiability checking can be done efficiently for feature models [MWC09].

Domain constraints, do not avoid feature combinations that are invalid because of an SPL's implementation. We currently achieve limited composition safety at the level of the implementation language because all features are compiled before runtime and syntax errors are avoided. Furthermore, we avoid errors due to implementation dependencies with additional *implementation constraints*. For example, when a feature's implementation references another feature, we can enforce a valid configuration with an according implementation constraint (which is technically not different from a domain constraint). In FeatureAce, we use an extended product line model that integrates domain and implementation constraints [SKR+08]. In combination with a type system that is aware of the feature model [AKGL10] we can ensure type-safety for dynamic binding without defining the feature selection at compile-time.

5.4 Compositional Overhead

We analyze the presented concept for dynamic feature composition with respect to resource consumption. We focus on binary size (footprint), consumption of working memory, and performance. To provide insights in the resulting compositional overhead we compare dynamic feature binding with static binding. We provide a brief analysis only to demonstrate the importance of the compositional overhead and to motivate the extension of the presented concepts in the next chapter. For evaluation, we use the Graph Product Line (GPL), a small SPL that implements graph data structures. The SPL supports colors, weights, and names of edges in individual features (features Color, Weight, Name). Class `Edge` implements edges between graph nodes.

5.4.1 Memory Consumption

In our current implementation, dynamic binding increases the size of an object to store pointers to the predecessor refinement of the refinement chain (i.e., the `super` pointer) and a pointer to the last refinement (i.e., the *object's* `this` pointer), as shown in Figure 5.13. To enable dynamic binding via the decorator pattern, all

[3] Boolean satisfiability problem.

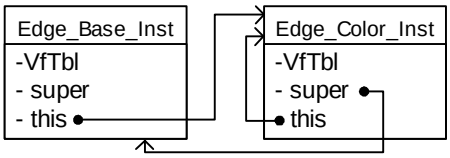


Figure 5.13: An object of class Edge consisting of two decorators for base implementation and feature Color.

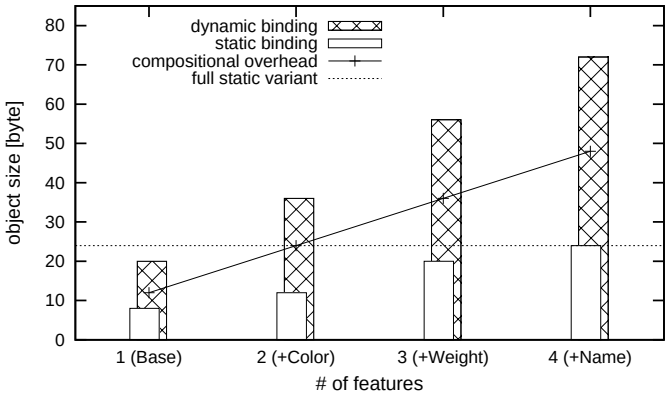


Figure 5.14: Size of an object of class Edge of different variants of the graph product line.

decorator objects have a virtual function table (`VfTbl` in Fig. 5.13). In a 32-bit environment each pointer increases the size of an object by 4 bytes per refinement. A proxy object (cf. Sec. 5.2.2) is optional for runtime adaptation and we do not include it in the following evaluation.

Hence, there is a linearly increasing *compositional overhead* of 12 bytes per refinement in case of dynamic composition. Dynamically composed objects are thus larger than their statically composed counterparts. In Figure 5.14, we depict the object size for class `Edge` of the GraphPL using dynamic binding (cross-hatched bars). The size of an object of class `Edge` increases linearly with an increasing number of features. For comparison, we also depict the size of the object when using static binding (white solid bars). If the actually needed features are not known at compile-time, more features have to be included when using static binding. This results in a constant object size of 24 bytes if we include all features in a static variant (dashed line in Figure 5.14). It causes a *functional overhead* when the features are not used. By contrast, we observe a variable object size of 20–72 bytes for dynamic binding. This causes a compositional overhead compared to the static variants (solid line).

Hence, the *functional overhead* may outweigh the compositional overhead: Edges of a minimal dynamically composed simple graph (configuration 1; only using module Base) are 17 % smaller than the edges of a statically composed graph with all features. This is important in case of a high number of edges. The differences in ob-

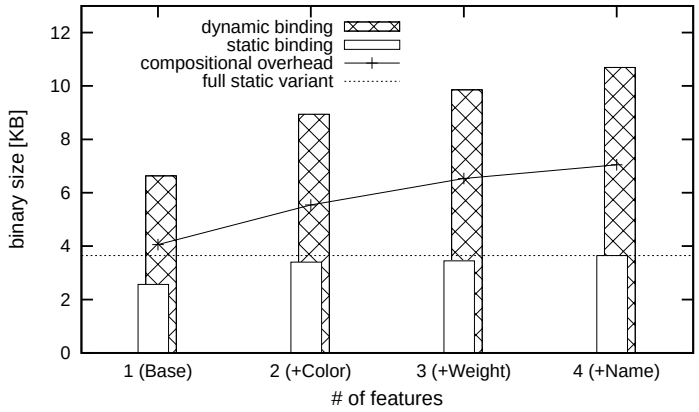


Figure 5.15: Binary size of different variants of the graph product line.

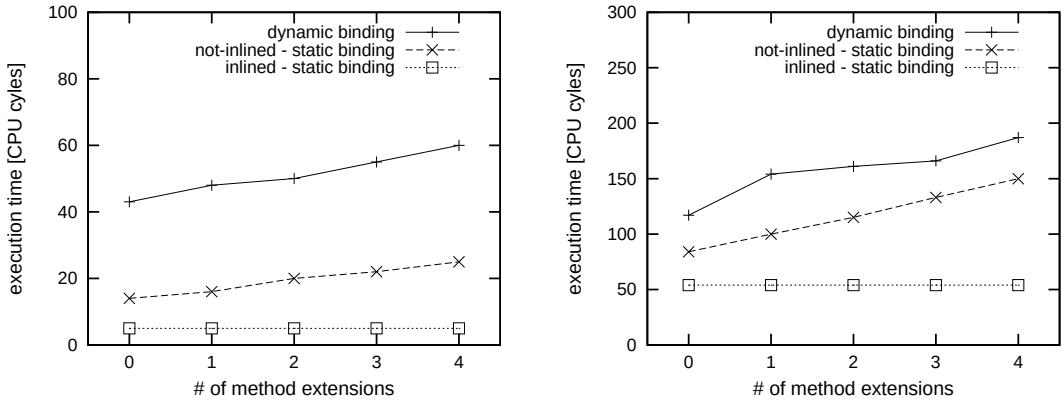
ject size highly depend on the actual size of the refinements. It results in quite worse memory consumption for our example in case of dynamic composition since we used small objects. The effect decreases with an increasing object size. When considering large refinements (with respect to the data size) and statically not known features, dynamic binding can achieve much better memory consumption than static binding (as demonstrated with configuration 1). This emphasizes the scenario-dependent differences in memory consumption for static and dynamic binding.

5.4.2 Binary Size

In Figure 5.15, we show the footprint of an application that uses different statically and dynamically composed variants of the graph SPL. The observed binary size of a statically composed variant is always smaller compared to dynamic composition. We observe a basic *compositional overhead* of about 4 KB which increases with an increasing number of features. The functional overhead for a statically composed full configuration is small compared to this (about 1 KB when only the base implementation is used). It cannot outweigh the compositional overhead as we observed it for memory consumption. However, the functional overhead increases when we add more features to the SPL.

5.4.3 Performance

Static binding may provide better performance than dynamic binding with decorators if a dynamically composed method consists of several fragments, one for each method refinement. Calling the method means an indirection for every method extension. When using static binding, the same method is not decomposed into different fragments and it may be inlined at the position where it is called. The overhead for dynamic binding thus highly depends on compiler optimizations and the hardware.



a) Frequently called methods. b) Single method execution.

Figure 5.16: Execution time of dynamically and statically composed methods for an increasing number of method extensions a) with and b) without caching.

In Figure 5.16, we show the number of CPU cycles required to execute different variants of a tiny method[4] with an increasing number of empty refinements using static and dynamic binding. In part a) of the Figure, we depict the execution time when the different method variants are frequently called in a running program. That is, most of the data (such as accessed member variables and pointers to decorators) is already loaded into the CPU cache. This is important for hotspot methods of a long-running software that are repeatedly called. In part b) we depict execution of the same methods without caching. This is important for software that is often restarted (e.g., a compiler) and methods that are less frequently called. For static binding, we depict execution time with and without inlining of method extensions. Execution of the base implementation takes between 5 (Figure 5.16 a) and 54 (Figure 5.16 b) CPU cycles with and without caching respectively. Execution time of the base implementation is depicted as *inlined - static binding*. Method extensions are empty and do not increase the execution time when using static binding and inlining; the execution time slightly increases when inlining is disabled.

In case of dynamic binding, the object is decomposed into decorators (one for each refinement). The resulting overhead for dynamic binding is at least 38 CPU cycles for frequently called methods. For methods that are only called once it is higher than 63 CPU cycles. In the diagrams, these are the differences between dynamic and static binding with inlining. In a real system, the average overhead will be between both extremes. For an increasing number of method extensions, the overhead increases by about 4 and 18 CPU cycles per method extension with and without caching respectively. For large methods that require thousands of CPU cycles the observed overhead can be ignored. However, small methods may degrade the performance of a program significantly as also observed for virtual methods in

[4]The method manipulates a member variable of a class (addition and multiplication of an integer).

C++ programs [AH96]. Furthermore, the increasing code size may also increase the number of cache misses due to an increased size of executable code and data (e.g., to store pointers for decorators). Hence, the practical relevance of the performance loss due to dynamic binding can only be estimated with a real case study, as we show in the next chapter.

5.4.4 Conclusion

Our evaluation shows that the chosen binding time influences the resource consumption of an SPL. The more we know about the functionality that is required at runtime the better we can optimize an SPL instance. That is, we can optimize an SPL instance by removing unneeded features and by choosing the correct binding time. The presented approach allows a programmer to do this at deployment-time per SPL instance without the need to modify the implementation manually.

The actual compositional overhead depends on the used programming language and the execution environment. Even though our implementation is not optimized with respect to resource consumption, we cannot completely avoid the overhead for dynamic binding and thus have to reduce it to an acceptable minimum. This is especially important in case of many small class refinements. In the next chapter, we extend the approach to integrate both binding times for a single SPL.

5.5 Related Work

In the following, we discuss related approaches that support dynamic binding with respect to aspects not discussed in Chapter 3. We present related work with respect to a combination of static and dynamic binding in the next chapter.

Our approach for dynamic binding is partially based on Delegation Layers [Ost02] which supports dynamic composition of features but currently lacks an implementation. Similarly, Object Teams support dynamic binding of *teams*, which can be used to represent features of an SPL [HMPS07]. We extend Delegation Layers by *generating* the enclosing classes that represent layers and thus reduce the complexity of an SPL's implementation. Furthermore, when clients use a single SPL instance only, we avoid the implementation overhead required for virtual types. In contrast to Delegation Layers and other approaches that support dynamic binding (cf. Sec. 3.2), we support safe composition of features by including the feature model into the running program. When combining this with static type checking, we can achieve type-safe composition at runtime.

Kegel et al. have shown how inheritance can often automatically be refactored into delegation [KS08]. They demonstrated that both approaches are quite similar and there is no major benefit when using one or the other, but delegation sometimes fails to replace inheritance. We use the decorator pattern and method forwarding to replace linear refinement chains which can also be implemented with inheritance as presented by Batory et al. [BLS98]. Since we do not replace inheritance in general, we do not observe the problems found by Kegel et al., e.g., when using abstract classes.

Instantiation of incomplete class refinements including abstract methods is possible because we generate methods that forward method calls to the next refinement in the refinement chain. If no refinement of a class implements the method an exception is thrown if it is invoked. Currently, we do not statically check if an abstract method is actually implemented by a concrete class instance. This can be enforced with a static type system for product lines [AKGL10].

5.6 Summary

We presented an approach that supports static and dynamic binding of features based on a single extension mechanism, i.e., class and method refinements. This allows a programmer:

- to reduce development effort when different binding times for an SPL are required because only a single implementation is needed,
- to simplify SPL development, because only a single implementation mechanism (i.e., method refinements) has to be learned and used,
- to improve reuse between different SPLs, because features can be reused in SPLs that require static or dynamic binding,
- to choose the binding time of an SPL later in the development process (e.g., per application scenario) without changing an SPLs implementation.

We have shown how client programs developed in FeatureC++ or in plain C++ can create SPL instances at runtime. When using multiple SPL instances, we support an emulation of virtual classes for generated C++ code. Finally, we support automated composition and composition safety by including the feature model in a generated dynamic SPL. Combined with static type checking the entire SPL, we can achieve type-safety for dynamic binding even for unanticipated configurations, which is in contrast to existing approaches for dynamic binding.

Due to FeatureAce, our customizable composition framework, we can choose at deployment time which composition capabilities are needed (e.g., which configuration mechanism). This increases the flexibility of feature binding and abstracts from details of the binding mechanism. However, the presented approach allows a programmer to choose the binding time only for the whole SPL and not for single features. This results in two major shortcomings of the approach:

Limited Flexibility: For some SPLs and application scenarios it is not possible to freely choose between static and dynamic binding. For example, static binding has to be used for features that are specific for a build or execution environment (e.g., choosing between two alternative implementations of a type) and when some language features of C++ are used (e.g., `sizeof`). There are also scenarios that do not allow us to use static binding. For example, we have to use dynamic binding when we do not know at deployment time which of two mutual exclusive features to use.

Increased Resource Consumption: The resource consumption of an SPL is not optimal when we have to use static or dynamic binding exclusively. This is caused by a *functional* or *compositional overhead* depending on the chosen binding time: Static binding results in a *functional overhead* when features are included in a program variant but are not used. By contrast, dynamic binding causes a *compositional overhead* and enables changes or extensions of a program after deployment.

We address both shortcomings in the next chapter with an approach that seamlessly integrates static and dynamic binding by choosing the binding time per feature.

6 A Generative Approach to Flexible Binding Times

This chapter shares material with the ASE Journal paper "Flexible Feature Binding in Software Product Lines" [RSAS11] and the VaMoS'11 paper "Multi-Dimensional Variability Modeling" [RSTS11].

The approaches we presented in the previous chapters allow programmers to choose between static and dynamic binding for an entire SPL only and not for single features. This is too inflexible for certain application scenarios and results in a *functional* or *compositional overhead*. Especially the compositional overhead limits applicability of purely dynamic binding. To overcome this limitation, we reduce the compositional overhead by combining static and dynamic binding.

In Figure 6.1, we illustrate scenarios for static and dynamic binding that are supported by different approaches for software composition. Features are shown as letters and generated binary code units are shown as ellipses. In the upper half of the figure, we show purely static (left side) and purely dynamic (right side) binding of features. Both mark the two ends of compositional approaches with respect to the supported binding time. For static binding, all features are composed into a single program (upper left) and for dynamic binding each feature module is transformed into a separate binary code unit. In the lower half of the figure, we show combined approaches. On the right side, we depict one way to combine both binding times as supported by most existing approaches. It allows a feature to be bound statically (features A, B, and C) or to be bound dynamically in a separate module (features D–I). The drawback of this solution is that a highly dynamic application requires many dynamically bound modules, one for each feature. When binding all features dynamically, this even results in purely dynamic binding with the same compositional overhead as discussed in the last chapter. On the lower left part of the figure, we depict the solution we propose to closely integrate both binding times: we combine multiple dynamically bound features into a single module if the features are always bound at the same time (e.g., D, E, F). The resulting *dynamic binding units* reduce the compositional overhead because communication within a binding unit does not require dynamic binding (e.g., using virtual methods or conditional statements). Hence, a single feature, such as feature E, is statically bound with respect to features D and F but dynamically bound with respect to the whole program. This solution is also achieved when manually developing components [LK06] but it requires that the components are planned before development.

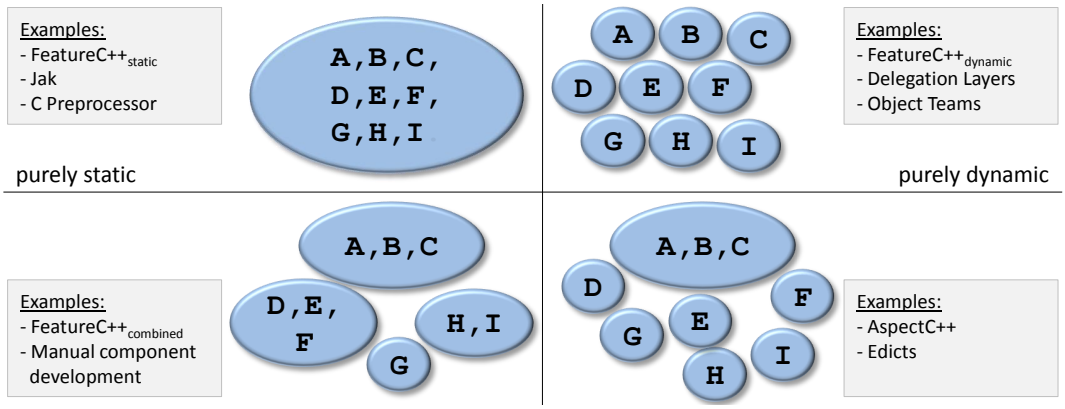


Figure 6.1: Approaches for purely static and purely dynamic feature binding (upper part) and two possibilities to combine both approaches (lower part). Features are shown as letters; binary code units are shown as ellipses.

Next, we present an approach for *generating* dynamic binding units from a set of user-defined features based on purely static and purely dynamic binding described in the previous chapters. We first introduce dynamic binding units and explain the general concept. We then formalize feature composition and present an extension of feature modeling approaches. Finally, we have a closer look at the code transformations that are needed to integrate static and dynamic binding.

6.1 An Overview: Dynamic Binding Units

We integrate static and dynamic binding by composing features with the same binding time into a single module: When multiple features are always bound together we merge these features at compile-time into a *dynamic binding unit*. Lee and Kang propose to develop dynamically bound components that correspond to a set of features, i.e., feature binding units [LK06]. This requires domain engineers to choose the features of a binding unit before development. Instead of manually developing binding units, we automate this process and generate them on demand at deployment time. That is, a programmer implements an SPL once and chooses the binding time per feature later. This means a two step composition process: First, we use static composition for features within a binding unit and, second, we compose dynamic binding units in the running program similar to component composition. The static composition process results in a *prebound* SPL consisting of a set of dynamically composable binding units, each of which consists of possibly multiple statically composed features. As a binding unit may correspond to a single feature or may include all features, purely dynamic and purely static binding are special cases of the presented approach. Binding units reduce the complexity and the overhead of dynamic binding since multiple features are bound simultaneously.

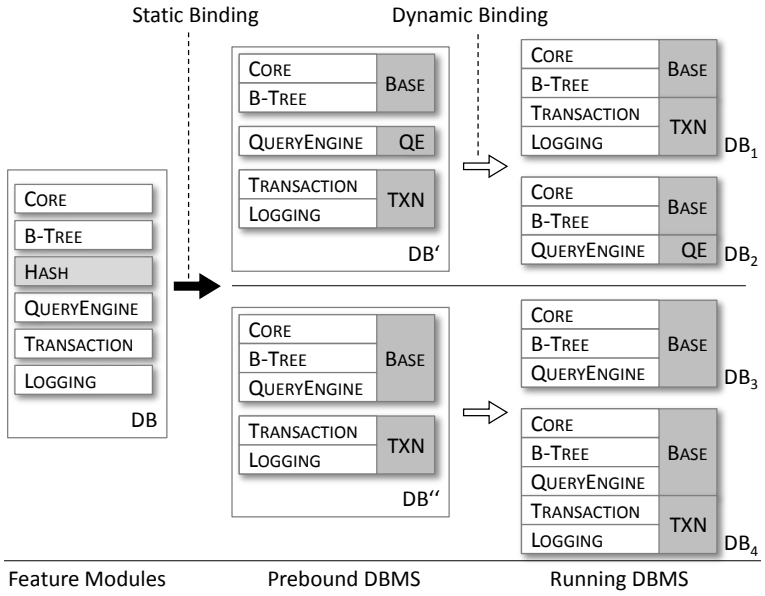


Figure 6.2: Two examples for static transformations (➡) of a DBMS product line resulting in prebound product lines DB' and DB'' and subsequent dynamic composition (⇨) resulting in running programs DB_1–DB_4. Feature Hash was not selected and is not included in any binding unit.

Generating Binding Units. In Figure 6.2, we show a possible scenario for generating dynamic binding units for a DBMS. DB' and DB'' denote two prebound product lines (i.e., not concrete products) after static composition. The prebound SPLs provide less variability than DB because the features of a binding unit can only be selected in combination. In DB', we assume that feature B-Tree is always required and we combine it with feature Core into a single binding unit Base. Similarly, Transaction and Logging are composed into binding unit Txn. Feature QueryEngine is assigned to another distinct binding unit QE. This is different in DB'', in which feature QueryEngine is not assigned to its own binding unit but added to binding unit Base. Feature Hash is not required and is thus not assigned to any of the binding units. From each prebound DBMS SPL we can create a number of different concrete DBMS instances (examples DB_1–DB_4) by dynamically composing the binding units according to a given configuration (i.e., a list of required binding units). Comparing DB_2 and DB_3, we see that both provide the same functionality but feature QueryEngine is bound dynamically in DB_2 and statically in DB_3, which leads to differences in flexibility and resource consumption.

Product Derivation. The product derivation process of our integrated approach can be divided into three steps: (1) configuration, (2) static transformation, and (3) dynamic composition. In the first step (filled arrow in Figure 6.2), a user se-

lects the potentially required features and assigns each feature to a binding unit. In the subsequent static transformation process, the compiler selects the required feature modules and generates dynamic binding units. The compiler also generates code for composing the binding units at runtime. There are two extremes: first, a single binding unit contains all selected features which results in a purely statically composed program without any code for dynamic binding. Second, each binding unit may contain only a single feature resulting in a purely dynamically composable SPL. Between these extremes (which mark the current state of the art) our extended approach supports any combination of static and dynamic binding.

After static transformation, the dynamic binding units are composed as described for the purely dynamic approach. This includes validation of the configuration before composing the binding units at runtime. This is done in the same way as we have shown for purely dynamic composition using a feature model (cf. Sec. 5.3.3). Since a feature of an SPL maps to multiple binding units, this is not possible by using the original feature model. In the next section, we present a way to transform a feature model according to the generated binding units. This allows us to use the transformed model for validating the composition also for the combined approach.

The described process consists of two configuration steps for static and dynamic binding. In general, we can have more composition steps and we may also use other binding times, such as binding at link time. In the following, we generalize the composition process and demonstrate how it can be included in a staged configuration process.

6.2 Staged Feature Composition

In FOP, a feature is implemented in a feature module (cf. Sec. 2.2.5). A dynamic binding unit can be seen as a *compound feature module* that is composed from many smaller feature modules. We call the process of stepwise feature composition *staged composition*. It corresponds to *staged configuration* of feature models described by Czarnecki [CHE04]. In the following, we first introduce *compound features*, which we use to represent *compound feature modules*. Second, we extend staged configuration to model staged composition.

6.2.1 Compound Features

We formalize feature composition by treating features as functions that modify other features or a base program [LBL06b, ALMK10]. Composition of a feature with another feature results in a *compound feature*, which is the source for a next composition step. In our case, a dynamic binding unit is a compound feature module that is bound in a dynamic composition process. We denote static feature composition with operator $\bullet$ and dynamic feature composition with operator $\circ$. This way, we can describe composition of programs DB_1 and DB_2 (cf. Fig. 6.2) as follows:

$$
\begin{aligned}
Base &= BTree \bullet Core && (6.1)\\
QE &= QueryEngine && (6.2)\\
TXN &= Logging \bullet Transaction && (6.3)\\
DB_1 &= TXN \circ Base && (6.4)\\
&= (Logging \bullet Transaction) \circ (BTree \bullet Core) && (6.5)\\
DB_2 &= QE \circ Base && (6.6)\\
&= (QueryEngine) \circ (BTree \bullet Core) && (6.7)
\end{aligned}
$$

In this example, equations 6.1–6.3 represent static compositions resulting in compound features (i.e., dynamic binding units) *Base*, *QE*, and *TXN*. Equations 6.4 and 6.6 represent dynamic compositions of compound features. Hence, a feature such as TRANSACTION is statically bound with respect to its binding unit *TXN* but it is dynamically bound with respect to the whole program.

Note, when using multiple composition steps we have to consider the order in which features are composed. The reason is that composition of feature modules is not necessarily commutative [ALMK10]. For example, when features TRANSACTION and BTREE extend the same method, it is usually important which method refinement is executed first. Hence, when changing the two dynamic units from Equations (6.1) and (6.3) to $Base = Transaction \bullet Core$ and $Log = Logging \bullet Btree$, dynamic composition results in a different program:

$$DB_{1'} = (Logging \bullet Btree) \circ (Transaction \bullet Core). \quad (6.8)$$

$DB_{1'}$ differs in its behavior from DB_1 if composition of $Btree$ and $Transaction$ is not commutative, which is the usual case. We consider this when combining static and dynamic binding using special code transformations as we describe in Section 6.3.

6.2.2 Staged Product Derivation

Staged composition and staged configuration are closely related. Staged configuration is a configuration process on a conceptual level that results in a set of (partial) SPL configurations with reduced variability, as shown in the upper part of Figure 6.3. The configuration steps usually do not correspond to composition steps. For example, a DBMS expert could decide which index to use in a DBMS (configuration step 1 in Figure 6.3) and another expert could decide whether transactions are needed or not (step 2). After applying these configuration steps, we can use a single code generation process to derive a product directly from the final configuration (step c). This process of staged configuration does not require staged composition. Alternatively, we can use several composition steps that correspond to the partial configurations (lower part of Figure 6.3) or use a combination of both. For example, we can use staged configuration step 1, generate the corresponding modules (b), and finally compose the modules (step 2'). However, staged composition is always used in combination with staged configuration. For example, for composition step 2' we need the corresponding configuration step 2.

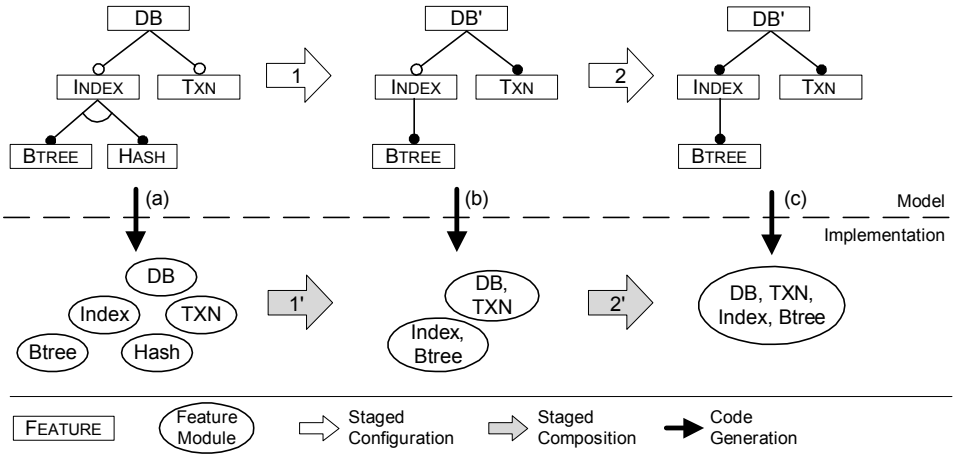


Figure 6.3: Staged product derivation using staged configuration at the model level and staged composition at the implementation level.

Staged composition can mean to merge multiple feature modules into compound feature modules. For example, we compose modules Index and Btree in composition step 1' of Figure 6.3. When we combine static and dynamic binding, composition step 1' may represent static composition and step 2' dynamic composition. The resulting compound module is a dynamically composed program. It is a large feature module dynamically composed from smaller feature modules. Compared to purely static composition, the program provides the same functionality but is composed from dynamic binding units. In general, we can have several composition steps with different binding times. For example, step one could be split into two static composition steps.

6.2.3 Staged Configuration and Compound Features

We represent multiple composition steps at the model level using staged configuration. This allows us to validate the composition process and to use the preconfigured feature model for subsequent configuration steps, such as dynamic composition at runtime. We transform the feature model according to the generated binding units, which results in feature model that includes only variability and commonalities among products that can be derived dynamically. To support arbitrary configuration steps we extend the steps described by Czarnecki et al. [CHE04] with *configuration constraints* (i.e., arbitrary propositional formulas) that are added to the feature model [RSTS11]. This allows us to represent the merge operation that yields compound features with an equivalence constraint between the merged features.

In the upper part of Figure 6.4, we depict an example for the model transformation of *DB* into *DB′* (cf. Fig. 6.2). We add constraints Core, Txn, Qe to represent the

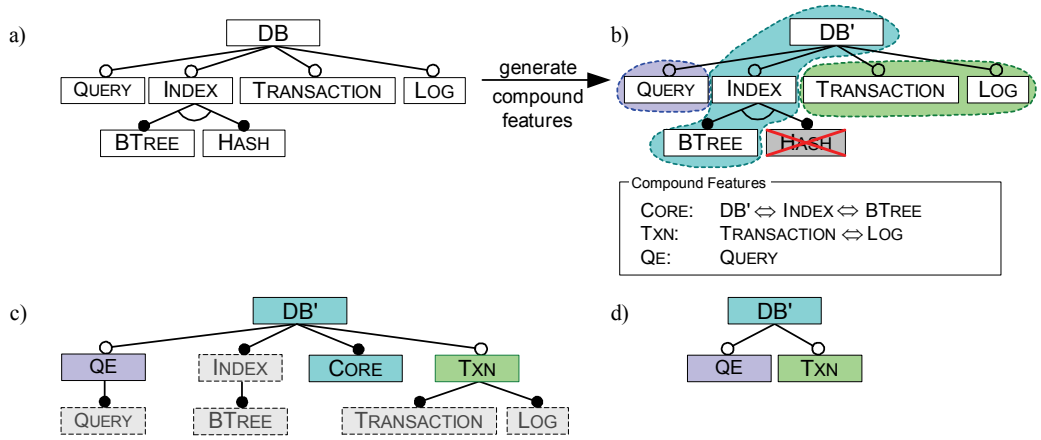


Figure 6.4: Transformation of a feature model when merging multiple features into compound features (CORE, TXN, and QE). In (b), constraints are used to represent the merge operation. Compound features are added in (c) and a refactoring is used to remove the equivalence constraints. Mandatory features have been removed for simplicity in (d).

compound features. Furthermore, we remove feature HASH, which was not selected for composition. The resulting feature model correctly represents only dynamic variability. It forces a user to either select all merged features of a compound feature or to select none of them. For example, features TRANSACTION and LOG can only be selected in combination due to constraint TXN.

We use the feature model transformation to validate the composition process: After transformation, we can check whether the resulting feature model is satisfiable, i.e., if there are valid variants. It also allows us to find invalid transformation steps by applying the transformations in a step-wise manner. For example, when we merge features BTREE and HASH the model transformation results in an unsatisfiable feature model because the created constraint BTREE ⇔ HASH conflicts with the XOR constraint between the features. Using a SAT solver, this can be checked efficiently for each configuration step [MWC09].

Feature Model Refactoring. The feature model shown in Figure 6.4 b is rather complex compared to the actual variability. Furthermore, it does not explicitly show the new compound features that are used for dynamic composition. To reduce the complexity, we can automatically refactor the feature model, which results in a simplified model, as shown in Figure 6.4 c. In the following overview, we describe the required refactoring steps. The whole process results in an equivalent feature model because each step is a refactoring that maintains the variability. We derived the refactoring steps by applying the refactorings described in [AGM+06].

1. In a first step, we remove dead features that cannot be selected. In our exam-

ple, this means that feature HASH is removed as it is an alternative to BTREE and cannot be selected. When removing a dead feature we also remove it from existing constraints to other features (e.g., replacing it with *false* in boolean constraints).

2. We remove a feature from the equivalence constraint of a compound feature when it is the ancestor of one of the other features. In turn, we have to mark the features on the path between both in the feature model as mandatory. For example, we mark features INDEX and BTREE as mandatory and remove the equivalence constraint for compound feature CORE. Mandatory features that have been part of an alternative group must always be selected. Hence, remaining features of the group cannot be selected, but these have already been removed in step 1.
3. In steps 3–5, we add the compound features and restructure the feature diagram. First, we create a new feature for each generated compound feature (e.g., feature QE in Figure 6.4 c). Each compound feature replaces one of the merged features. Usually, the compound feature should replace the feature that is nearest to the root. The replaced feature is added as a mandatory child since both have to be selected at the same time. For example, we insert compound feature QE above QUERY. If one of the merged features is the root of the tree the compound feature may also be added as a child of the root to avoid a different name for the root (cf. feature CORE in Figure 6.4 c).
4. Other merged features including their whole subtrees are moved to the corresponding compound feature as mandatory child features (e.g., feature LOG in Figure 6.4 c). Additional constraints are added to maintain the relationships between the moved features and their former parent features and siblings. In our example, we create the constraint LOG ⇒ DB' because feature TXN was added as a parent of TRANSACTION.
5. Finally, we remove constraints that are not needed. Since the merged features are mandatory children of their compound feature we remove the remaining equivalence constraints that have been used to represent merged features as described in step 2. Furthermore, we can remove some constraints that have been added in step 4. For example, we remove constraint LOG ⇒ DB' because DB' is an ancestor of LOG.

After refactoring the feature model the remaining variability is easier to recognize because it is not hidden in constraints. The merged features can be removed from the feature model or tool support can be used to suppress visualization of mandatory features. When removing the inner features of a compound feature, the constraints have to be updated by replacing the removed features with their compound feature. However, the original merged features may still be needed for further operations on the feature model. For example, rules for adaptations of a feature model at runtime or constraints defined independently (e.g., by a third party) may reference the original features.

In general, merging features corresponds to adding equivalence constraints to the feature model. The variants of the feature model shown in Figures 6.4 b and c are equivalent and both can be used for further compositions or reasoning about the feature model. The difficulty of the described refactoring comes from finding a representation in the notation of feature diagrams that is easier to understand for a user than propositional formulas as in Figure 6.4 b. Hence, there may be more possible simplifications that depend on the structure of the feature tree.

6.2.4 Composition Safety

We use the transformed feature model to achieve safety for composition of binding units. By statically type checking the entire SPL [AKGL10], we could even provide a type-safe composition mechanism for binding units. By contrast, customizable components that implement multiple features (e.g., implemented with the C preprocessor or AOP) do not support type-safety with respect to all possible configurations.

For clients that use an instance of an SPL, we support a limited form of virtual types as described in Section 5.3.1. However, a client has to use the binding units for manual SPL instantiation. For automatic instantiation, it is also possible to use the SPL features, which are mapped to the required binding units.

6.3 Using Code Transformations to Support Different Binding Times

For a proof of concept, we integrated the presented combination of static and dynamic binding into the FeatureC++ compiler. In the following, we give an overview of the code transformations that we use to combine both types of composition at the level of object-oriented classes.

6.3.1 Generating Binding Units

When generating dynamic binding units, the FeatureC++ compiler transforms a class (defined as several class refinements) of an SPL into dynamically composable class fragments. The generated fragments correspond to the defined binding units.[1] They are generated from the refinements of a class in two steps: First, we merge refinements belonging to features of the same binding unit into a single class (static composition) and, second, we generate code for dynamic binding of composed classes using the decorator pattern as described for dynamic composition in Section 5.2.1. In Figure 6.5, we show an example for the generated classes corresponding to the binding units of DB' (cf. Fig. 6.2). The dynamically composable class `DB` consists of an interface (`DB`), an abstract decorator (`DB_Decorator`), and three concrete decorators (`DB_Base`, `DB_QE`, and `DB_TXN`). Code of multiple refinements is statically

[1] Dynamic binding units are stored in the binary of an application or in extension libraries. Currently, we support Windows DLLs.

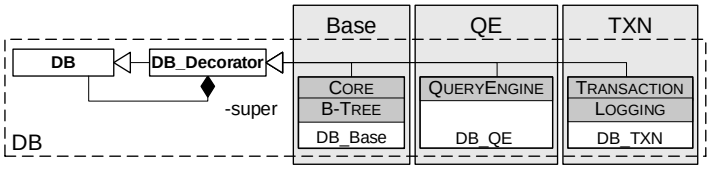


Figure 6.5: Compound class `DB` (dashed box) after static composition and transformations to enable dynamic binding. Generated decorators are shown as white boxes within light-gray binding units. Code of refinements is shown as gray boxes in decorators.

composed into the concrete decorators. For example, we merge refinements defined in modules CORE and B-TREE of class `DB` into decorator `DB_Base`. The decorators are combined at runtime according to the selected dynamic binding units. For example, we compose `DB_Base` and `DB_TXN` to yield DB_1 as shown in Figure 6.2.

The code transformations are basically a combination of the transformations described for purely static and purely dynamic binding. However, they differ in several ways and we describe important differences next.

Storing SPL Context. Class instantiation in a dynamically composed program requires to create an object that corresponds to the configuration of an SPL instance. In FeatureC++, we use objects of dynamically composed *feature classes* (cf. Sec. 5.2.1) to represent SPL instances. For example, instance DB_1 (cf. Fig. 6.2) includes binding units BASE and TXN of DB'. It is dynamically composed from objects of two feature classes (one for each binding unit). When a class instance is created, the configuration of the SPL instance defines which decorators to use for the newly created object. Because there may be more than one active SPL instance within a program, we need to know which SPL instance to use when creating an object. For that reason, we store a reference to the corresponding SPL instance within each object. For example, when creating an instance of class `DB` (cf. Fig 6.5), the SPL instance DB_1 defines the required binding units (BASE and TXN) and thus the configuration of class `DB`.

For statically composed classes, this information is not needed because the type of a class is determined statically and does not change according to a dynamically changing SPL instance. For example, the type of a class does not change if it is only refined in features that are part of the same binding unit. There is no runtime variability for such a class and we thus do not need to know which SPL instance to use for creating objects of the class.

When combining static and dynamic binding, we have to evaluate whether an object (directly or indirectly) creates instances of dynamically composed classes or not. This also applies to objects of statically composed classes that create instances of dynamically composed classes. For example, a class `QueryEngine`, which is statically composed because it is only part of binding unit QE, has to store a reference

to its SPL instance if it creates objects of dynamically composed classes.

Optimizing Resource Consumption. Our approach and its current implementation do not provide an optimized solution for every application scenario. There are several possible optimizations of the code transformation process to decrease the compositional overhead caused by dynamic binding. As a first step for optimization, we distinguish between using a single SPL instance and using multiple instances at the same time. For example, we may use differently configured instances of a DBMS. The FeatureC++ compiler optimizes the generated code when using only a single SPL instance. In this case, the compiler does not use SPL context variables in each refinement but uses a global reference for accessing the SPL instance. This reduces memory consumption and execution time.

There are further possible optimizations but we often observe a tradeoff between reducing memory consumption and increasing performance, which has to be further analyzed. For example, the code generated by FeatureC++ is not optimized with respect to memory allocation. We could allocate a single block of memory for all decorators when creating an object of a dynamically composed class instead of multiple blocks, one for each decorator. This allows us to reduce the object size by removing the `super` and `self` pointers. Instead, we can compute the memory addresses of the super object and the compound object for each compound class at instantiation time, as it is also done for inheritance by C++ compilers [Lip96]. Hence, the memory consumption of small dynamically composed objects could be reduced to about one third. However, this solution is only better suited for SPL configuration at load-time. Using this approach for reconfiguration at runtime (e.g., adding a new refinement to an already existing object) means that we have to reallocate the whole object when its size increases (i.e., when a new feature is loaded). This may highly increase the time required for adaptation. Hence, such optimizations are usually well suited for a particular application scenario only.

Commutativity of Method Refinements. Since application of method refinements is usually not commutative, we have to ensure that the execution order of method refinements does not change due to the combination of static and dynamic binding. We depict an example in Figure 6.6. Method `Put` of class `DB` is refined in features Logging and Transaction. Both method refinements have to be executed bottom-up: first the transaction code has to be executed (Line 12) and afterwards the logging code (Line 6). When we statically compose the Core implementation and feature Transaction into a single binding unit (Lines 1–12 in the generated code in Figure 6.7) and feature Logging into a different binding unit (Lines 13–18), then dynamic composition of the binding units results in an invalid program because the execution order of the method refinements changes.

To avoid this, we generate hook methods [AKB08]. For example, we generate method `Put_hook` as shown in Lines 4–6 in Figure 6.7. The hook is called in Line 10 instead of method `Put_Core`. It is overridden by feature Logging to execute the

Feature CORE

```
1  class DB {
2    bool Put(Key& key, Value& val) { ... }
3  };
```

Feature LOGGING

```
4  refines class DB {
5    bool Put(Key& key, Value& val) {
6      ... //logging specific code
7      return super::Put(key,val);
8    };
9  };
```

Feature TRANSACTION

```
10 refines class DB {
11   bool Put(Key& key, Value& val) {
12     ... //transaction specific code
13     return super::Put(key,val);
14   };
15 };
```

Figure 6.6: FeatureC++ source code of class `DB` with method `Put` extended in two features.

logging specific code before executing the extended method (Line 16).

Feature Interactions. Feature interaction code (a.k.a., derivatives) is only needed when a set of features is used in combination. When using static feature binding, the interaction code is composed with other feature modules into a single program. By contrast, the same code must be bound as a separate module when using dynamic binding (cf. Sec. 5.2.1). Hence, when combining both binding times, we must also ensure that existing interaction code is only executed when the corresponding features are active. This results in two possible scenarios:

1. All features of an interaction module are part of the same binding unit. In this case the interaction code is statically composed into this binding unit.

2. The interacting features are spread across different binding units. In this case, we generate a dynamic interaction module for each combination of binding units that interact. For example, we generate an interaction module when a feature of binding unit A interacts with a feature of a binding unit B. This interaction module includes all interaction code of binding units A and B (i.e., possibly code of multiple feature interactions). When there are higher order interactions (i.e., between more than two features) that are spread across more than two dynamic binding units, then this results in higher order dynamic interaction modules between these binding units.

Compared to purely dynamic binding, this also means a reduction of the number of interaction modules. When assuming a quadratic increase of interaction modules for an increasing number of modules then the use of dynamic binding units instead

Binding UnitBase

```
1  class DB_Base {
2    bool Put_Core(Key& key, Value& val) { ... }
3
4    bool Put_hook(Key& key, Value& val) {
5      return Put_Core(key,val);
6    }
7
8    bool Put(Key& key, Value& val) {
9      ... //transaction specific code
10     return Put_hook(key,val);
11   };
12 };
```

Binding UnitTXN

```
13 class DB_Logging {
14   bool Put_hook(Key& key, Value& val) {
15     ... //logging specific code
16     return super->Put_hook(key,val);
17   };
18 };
```

Figure 6.7: Generated C++ code of class `DB` with a hook for method refinement.

of purely dynamic binding means a quadratic reduction of the number of dynamic interaction modules.

Crosscutting Features. Feature interaction code is often needed for crosscutting features: When an optional feature cuts across other optional features we have to decompose the crosscutting feature into one interaction module for each feature interaction. For example, the transaction management of Berkeley DB cuts across many other features which results in several interaction modules. When binding a crosscutting feature dynamically, we have to create a dynamically bound module for each interaction. However, when binding the crosscutting feature statically, the code of all individual feature interactions is composed with the corresponding features into their binding units. Hence, we reduce the number of interaction modules and also the overhead for dynamic binding.

Due to the generated interaction modules, a compound feature maps to multiple binding units. Since multiple SPL features map to a single compound feature, there is an m-to-n mapping of SPL features to binding units. This is much the same as implementing an SPL with components: a component implements multiple features and crosscutting features are spread across multiple components resulting in an m-to-n mapping of features to components.

6.4 An Evaluation of Dynamic Binding Units

By means of two case studies we demonstrate the applicability of our approach. We evaluate the influence of different sizes and different numbers of dynamic

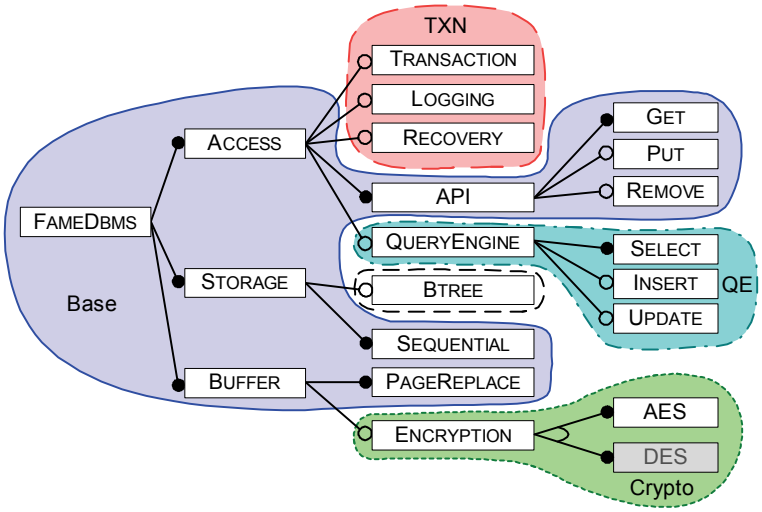


Figure 6.8: Feature diagram of FAME-DBMS with binding units Base, TXN, Btree, QE, and Crypto. Binding unit Crypto consists either of feature AES or DES. In our evaluation, we use feature AES.

binding units on resource consumption when using dynamic binding at load-time. For our evaluation, we use two product lines developed with FeatureC++. The first SPL is FAME-DBMS, a DBMS product line for resource-constrained environments [RALS09]. The second SPL is NanoMail, a customizable e-mail client. The source code of both product lines is available on the web.[2] We present the results for both SPLs and discuss the reasons for the characteristics we observed.

6.4.1 Defining Binding Units

FAME-DBMS. FAME-DBMS is an embedded DBMS (i.e., it is embedded into an application as a library). It was developed for devices with limited resources using static feature binding. In our case study, we use dynamic binding to achieve extensibility and optimized resource consumption. In Figure 6.8, we depict an extract of the feature model of FAME-DBMS and binding units used in our evaluation. We show only features that are relevant for our case study and omit features that are always statically bound such as operating-system-related features. In its current version, FAME-DBMS consists of 81 features with 14 200 lines of code (LOC).

For analyzing the influence of dynamic binding on resource consumption, we compare different variants of FAME-DBMS that use the same 44 features but we organized the features in different binding units. The selection of features per binding unit is shown in Figure 6.8. It corresponds to configuration 5 in the following analysis. We describe the rationale behind the definition of the sample binding units in

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/dynamic/

the following overview:

- Binding unit BASE represents a basic DBMS that consists of an API for storing and retrieving data. It can be used without additional binding units and provides high performance due to purely static binding.

- Binding unit TXN provides transactional access to the database. Since features TRANSACTION and RECOVERY require feature LOGGING, we merge all three features into a single binding unit.

- QE is a customizable query engine that supports a subset of SQL by statically composing only the required SQL features. In our implementation, dynamic composition of SQL features is hard to achieve. The reason is that we statically compose the SQL grammar from multiple features. We then generate the SQL parser from this composed grammar at compile-time. This demonstrates that purely dynamic binding is not always possible without increasing the development effort significantly.

- CRYPTO is a binding unit for data encryption and decryption. Customization of the encryption algorithms (e.g., AES or DES) is done statically. This means that we can exchange the encryption algorithm within the binding unit without modifying the remaining DBMS. We may also provide two different CRYPTO binding units, one with feature AES and one with DES. Moreover, a customer may provide an own encryption algorithm. Defining one binding unit for the DES and AES features would also be possible but in our case the ENCRYPTION feature abstracts from implementation details of the algorithm resulting in a small and uniform interface.

- Finally, binding unit BTREE provides efficient data access via a B^+tree index structure. In large DBMS there may be a number of different alternative index structures. Using a single binding unit per index structure allows us to activate only those that are needed for efficiently accessing the data.

NanoMail. NanoMail is an e-mail client SPL with 25 features and 6 200 LOC. It comprises different e-mail applications, from a simple MailNotify application that only notifies a user if there is unread mail, up to a full mail client with mail storage in a database. Similarly to FAME-DBMS, we compare variants with equal functionality (using 23 features) and varying binding units. For dynamic binding, we defined the binding units DB and CLAMAV, which provide mail storage in a database and virus filtering. Furthermore, for analyzing the impact of fine-grained dynamic customization, we provide e-mail filters that are used like plugins. Each filter is loaded as a single binding unit and users can add as many filters as needed. To analyze the influence of a large number of binding units, we generate several mail filters and measure the effect on the program's startup time.

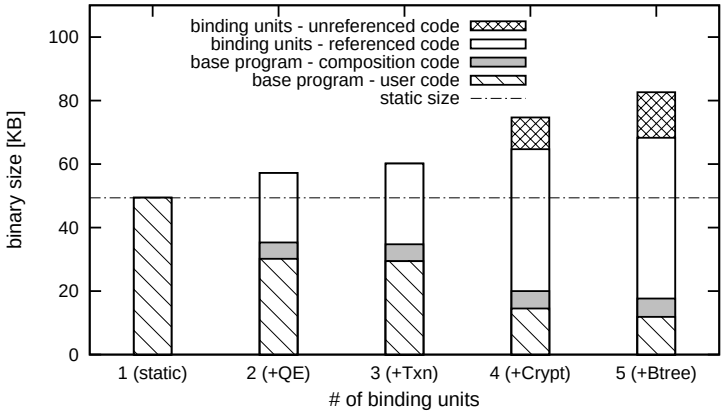


Figure 6.9: Binary size (base program and dynamic binding units) of five FAME-DBMS variants with an equal feature selection and an increasing number of binding units.

6.4.2 Resource Consumption

In the following, we analyze the resource consumption of different FAME-DBMS and NanoMail variants depending on the binding units used. We compare binary size, working memory usage, and performance of a varying number of binding units but we use always the same features.[3] Our aim is to identify how to combine static and dynamic binding to optimize a program with respect to functional and compositional overhead.

In Figures 6.9–6.14, we depict the results of our analysis for five configurations of FAME-DBMS and three configurations of NanoMail. In configuration 1, all features are statically bound and compiled as a single binary. In each of the configurations 2–5 an additional binding unit (e.g., QE, TXN, CRYPTO, BTREE for FAME-DBMS) is extracted from the base binding unit and compiled as a distinct dynamically linked library (DLL). In the following, we analyze binary size, memory consumption, and performance of both SPLs. We distinguish between compositional and functional overhead (cf. Sec. 3.1.2) for each analyzed property.

Binary Size

Using the binary size of FAME-DBMS, we first describe how we calculate the functional and compositional overhead. Since the functional overhead depends on the features actually used, we provide numbers for the *maximal possible* functional over-

[3] For our evaluation, we used an Intel Core 2 system with 2.4 GHz and Windows XP. For compilation and linking, we used the Microsoft C/C++ compiler v13.10.3077 and Incremental Linker v7.10.3077 (Visual C++ 2003). We used compiler optimization flag /O2 (i.e., /Og/Oi/Ot/Oy/Ob2/GS/GF/Gy). We linked dynamically against Microsoft's C++ runtime library and removed unreferenced functions and data with linker flag /OPT:REF.

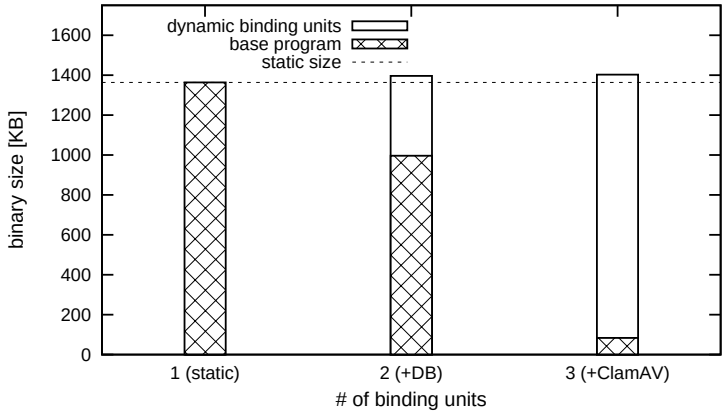


Figure 6.10: Binary size for three different variants of NanoMail with varying sets of binding units.

head. That is, we compare a static variant including all features with the minimal dynamic variant without additional binding units. The binary sizes of the configurations 1–5 of FAME-DBMS are shown in Figure 6.9. The values represent executable code and static data stored in the binary files. They do not include other libraries. For configuration 1, we generated a single binding unit including all features and five binding units for configuration 5. From configuration 1, we can only derive a single variant with a binary size of about 50 KB. Comparing configurations 1 and 5, we observe the following compositional and functional overhead:

- Comparing a complete variant of configuration 5 (83 KB) with configuration 1 (49 KB), we observe a *compositional* overhead of about 40 % (83 KB - 49 KB = 34 KB). That is, 40 % of the code from configuration 5 is overhead for dynamic binding.
- Comparing configuration 1 (49 KB) and the smallest variant of configuration 5 (18 KB), we observe a maximal possible *functional* overhead of about 64 % for configuration 1 (49 KB - 18 KB = 31 KB). That is, up to 64 % of the code of configuration 1 may not be used for a particular task (e.g., basic data storage and retrieval without using SQL queries and other features). The overhead depends on the number of features that are actually used at a particular point in time. It is zero when all features are really in use. This underlines that a configuration highly depends on the application scenario. To reduce the binary size, we have to avoid any features that are not used and reduce dynamic binding to a minimum.

In our case studies, we observe an increasing compositional overhead for an increasing number of binding units. Especially when a binding unit extends many classes the effect is very strong. It is quite strong for FAME-DBMS (up to 40 %, cf. Figure 6.9) and very weak for NanoMail (< 4 %).

The high relative overhead of 40 % for FAME-DBMS is mainly caused by its small binary size; the absolute overhead is 33.2 KB. The code of the composition infrastructure makes up 21 KB (25 % of the program size): About 5 KB generic code for dynamic binding (i.e., for loading and composing binding units; *base program - composition code* in Figure 6.9) and additionally between 3 KB and 5 KB overhead per binding unit (i.e., binding unit specific composition code). The remaining overhead of 12 KB (15 % of the program size) is caused by the binding units CRYPTO (9 KB) and BTREE (3 KB). Reasons are missing compiler and linker optimizations when dynamic binding is used. In Figure 6.9, we depict this unused code as *binding units - unreferenced code*. For example, a method for calculating a hash sum with a binary size of 7.5 KB is not used in our FAME-DBMS benchmark application. The linker removes the method from the statically composed variant because it is never called. The same method cannot be removed from binding unit CRYPTO because the compiler does not know whether it is required by another binding unit or not. Hence, dynamic binding may cause a functional overhead as well. This overhead is not caused by entire unused features but by unreferenced methods.

The possible functional overhead in static variants of both SPLs is very high (64 %–94 %). The reason is that a large fraction of the binary code belongs to optional features. Increasing the use of dynamic binding usually reduces this overhead. However, also insufficient customizability due to large binding units can cause a functional overhead when not all features of a binding unit are used.

Both kinds of overhead can be reduced by adjusting the binding units. That is, an application engineer has to analyze the functional and compositional overhead per application scenario to find the optimal tradeoff. When always using many of the binding units, the benefit of dynamic binding with respect to resource consumption decreases. For example, the binding units TXN and BTREE in FAME-DBMS cannot significantly reduce the functional overhead but they increase the compositional overhead significantly. The size of the base program is nearly the same in the configurations 4 and 5, but the additional binding unit BTREE increases the overall size by 12 %. That is, if we bind the features of the binding units TXN and BTREE statically (i.e., removing configurations 3 and 5), we do not cause a major functional overhead but can reduce the compositional overhead significantly. Hence, the application engineer has to decide whether this flexibility is really needed by taking the resulting compositional overhead into account.

Memory Usage

The memory usage of a program depends on allocated memory but also on the size of the binary program code that is loaded into memory. For FAME-DBMS, we could not measure any functional overhead of allocated memory because the memory is needed mainly for the data buffer of the DBMS, which is independent of the feature selection. Further features do not allocate a significant amount of additional memory. The functional overhead thus only depends on the binary program size and dynamic binding cannot reduce the memory consumption (cf. Figure 6.11). In NanoMail,

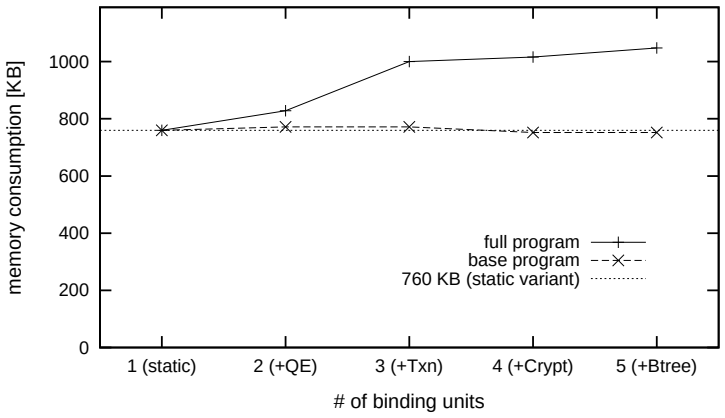


Figure 6.11: Comparison of working memory usage of five FAME-DBMS variants with an equal feature selection and an increasing number of binding units. *Full program* variants include all dynamic binding units. *Base program* variants only use the base program without loading additional binding units.

there is a functional overhead of about 28 % (cf. Figure 6.12), which is mainly caused by the virus filter. Comparing the mail-notify application (1.3 MB memory consumption; not shown in Figure 6.12) and a full mail client (9.8 MB), we observe a large functional overhead. This overhead is caused by including all features in the full mail client, which are not needed for a simple mail notification program (e.g., loading e-mails).

The compositional overhead of allocated memory especially increases if a program creates a large number of small objects that are dynamically composed. The reason is that the size of a class instance increases for each binding unit that crosscuts the class. This overhead is very high for small objects. The size S_{dyn} and the overhead O_{dyn} of a dynamically bound object can be calculated with the following formula:

$$S_{dyn} = S_{data} + O_{dyn} \quad (6.9)$$

$$O_{dyn} = 12(n_{BU} + 1) \quad (6.10)$$

S_{data} is the size of the object with static binding. n_{BU} is the number of *loaded* binding units that crosscut the class. The constant 12 represents the number of bytes a binding unit requires to store a self pointer, a super pointer, and a pointer to its virtual function table (each pointer has a size of 4 byte). The constant 1 represents the additional proxy that we use to enable reconfiguration at runtime. The proxy could be removed for SPLs that are not reconfigured once they are instantiated. Overall, the size of an object increases by 12 byte to enable dynamic binding and linearly increases by 12 byte for each additional binding unit.

In FAME-DBMS, the compositional overhead of 288 KB (38 %, cf. Figure 6.11)

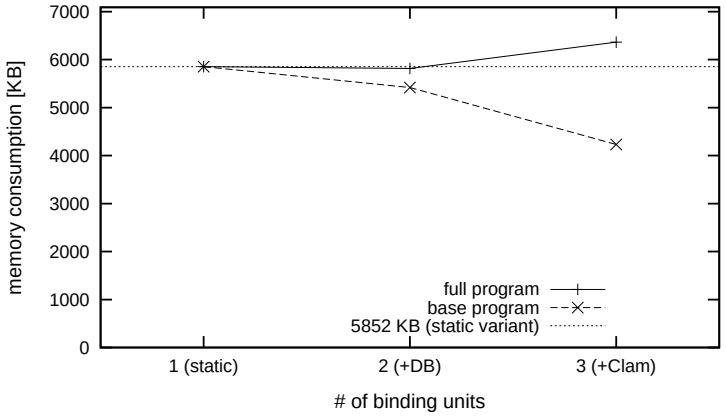


Figure 6.12: Consumed working memory in NanoMail for variants with different binding units.

is mainly caused by the binary program code and the overhead for loading binding units because there are only a few objects that are dynamically composed. The compositional overhead in NanoMail is 528 KB (9 %, cf. Figure 6.12).

Large binding units increase the functional overhead if some of their features are not used. The compositional overhead in FAME-DBMS is more important than the functional overhead; the opposite is true for NanoMail. The differences between the SPLs show that there is no general solution and there is space for domain-specific optimizations.

A binding unit also increases memory consumption due to the compositional overhead of its binary size (cf. Sec. 6.4.2) because the executable code is loaded into memory. We analyzed the overhead for loading a large number of binding units by adding several e-mail filters to NanoMail (up to 60 mail filters).[4] The results are shown in Figure 6.13. Besides a general overhead for dynamic binding (transition from 0 to 1 filters), we observe a linear increase of 21 KB per filter (i.e., per binding unit). For binding units that consume a small amount of memory, this is a large overhead. By generating one binding unit for multiple filters (i.e., merging the filters), this overhead can be avoided. For binding units that consume a large amount of memory, such as the virus filter in NanoMail, dynamic binding causes an acceptable overhead of only about 4 % compared to the memory consumption of the binding unit.

[4] We generated empty filter stubs to measure only the overhead for dynamic binding.

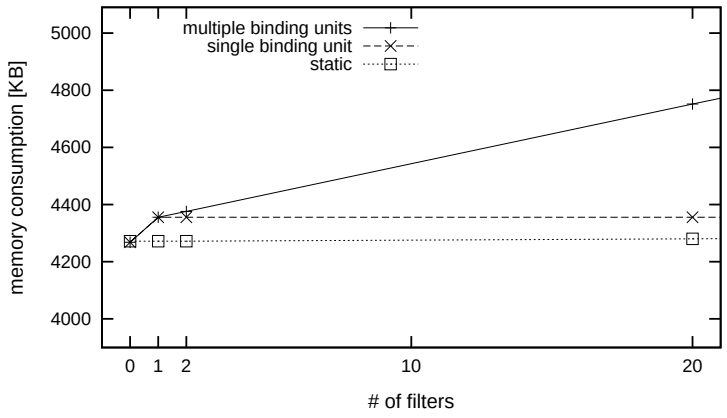


Figure 6.13: Consumed working memory of NanoMail with an increasing number of mail filters with static composition, dynamic composition with a single binding unit for all filters, and purely dynamic composition (i.e., one binding unit for each filter).

Performance

We measured the performance of FAME-DBMS using a benchmark for reading and writing data.[5] As shown in Figure 6.14, the performance decreases with an increasing number of binding units. Comparing dynamic variants with purely static binding, we observe a performance reduction between 5 % (2 binding units) and 28 % (5 binding units). The reason for this increasing compositional overhead is missing inlining of methods and more indirections for method calls compared to static variants. Both are caused by generated code for dynamic binding: composition of classes at runtime is achieved with virtual methods in decorators, which add an indirection and hinder inlining of method extensions.

In FAME-DBMS, 100 % of method refinements are inlined when using static binding. This decreases to about 95 % for configuration 2 and further to 86 % for configuration 5. Each method refinement that is not inlined is replaced by a virtual method and thus decreases performance. Binding unit BTREE substantially increases the overhead (cf. configuration 5 in Figure 6.14). The reason is that it refines methods that are invoked multiple times for a single read or write operation. Hence, we should create only a distinct binding unit for the Btree if this flexibility is really needed (e.g., when we have to decide at runtime which kind of index to use).

Static and dynamic binding may also affect the startup time of a program for loading binary code from DLLs and for initialization of unused code. Due to the fairly small binary size of binding units, we observe only a slightly increased startup time. The compositional overhead for loading binding units is about 30 ms per additional

[5] We used random key-value-pairs for reading and writing 10.000 records of type string via the B-Tree index.

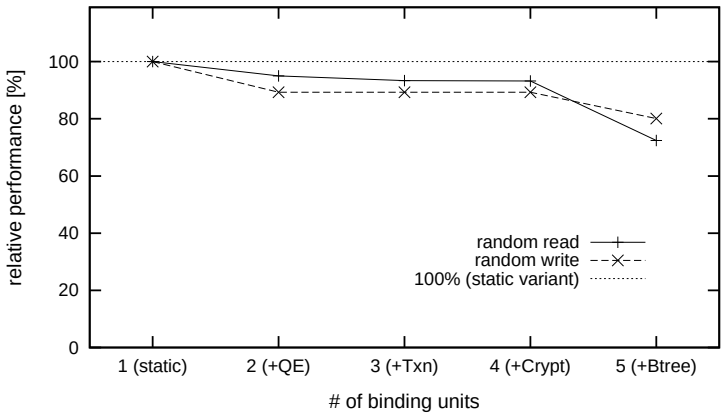


Figure 6.14: Comparison of benchmarks for reading and writing of different variants of FAME-DBMS. The relative performance is shown. 100 % means about 3.0 Mio queries / s for reading and 0.8 Mio queries / s for writing.

binding unit. We observe a functional overhead (intitialization code of features) of about 2 s for the largest binding unit in NanoMail (the CLAMAV virus filter). Hence, the compositional overhead with respect to program startup is very small and can be ignored in many application scenarios. By contrast, the functional overhead for initialization of a binding unit may be important for application scenarios that require restarting a program frequently.

To summarize, the influence of dynamic binding is quite high when a feature refines frequently called methods. The performance degradation can be caused by a high number of method extensions (e.g., in many binding units), but also by a few refinements of performance critical methods as shown for the BTREE feature in FAME-DBMS. Again, the best size for a binding unit has to be determined per SPL and application scenario. Merging binding units can remove dynamic method refinements. The load-time of a program can only be reduced significantly if the execution of complex initialization code can be avoided or if large parts of a program do not have to be loaded at startup. The number of binding units is usually not important. For example, 30 binding units result in an overhead of about one second for starting the programs of our case study.

6.5 Discussion

In the following, we discuss the results of our evaluation and analyze how customizability and SPL development is influenced by our approach. Finally, we derive a guideline for building SPLs that support static and dynamic binding.

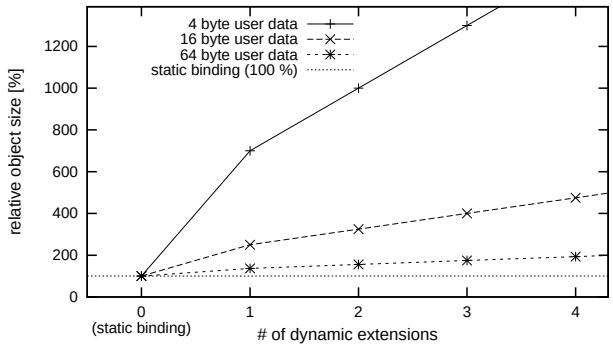


Figure 6.15: Relative size of three objects with a data size of 4, 16, and 64 bytes for an increasing number of dynamic extensions (i.e., crosscutting binding units).

6.5.1 Resource Consumption

Our evaluation has shown that, depending on the binding time, a compositional and functional overhead occurs in a running program with regard to binary size, memory consumption, and performance. The compositional overhead caused by a binding unit depends on its entanglement with other binding units. That is, method calls across the boundary of a binding unit (i.e., via its interface) introduce an execution time overhead. The interface of a binding unit consists of virtual methods to enable dynamic binding. This hinders method inlining, introduces indirections, and increases the size of generated code as well as the size of objects in a running program. Hence, a binding unit should contain feature sets that are used in combination. For example, the effect on memory consumption is very high when allocating a large number of small objects. In Figure 6.15, we depict the computed relative size (cf. Equation 6.9) of three different objects with 4, 16, and 64 bytes user data with an increasing number of dynamic extensions (i.e., dynamic binding units that crosscut the object). For an object with 4 byte user data, two dynamic extensions increase the object size by a factor of 10. If such objects are the main cause of memory consumption of a program then the memory consumption also increases by a factor of 10. For larger objects, this effect is much smaller. Combining static and dynamic binding reduces the number of dynamic binding units and can thus highly decrease the memory consumption. By merging dynamic binding units the overhead can be reduced. For example, merging all dynamic binding units of Figure 6.15 into a single binding unit reduces the memory consumption from 64 to 28 bytes for an object with 4 bytes user data.

However, large binding units introduce a potential functional overhead due to features that are not used. Splitting binding units can reduce the functional overhead, but we have shown that this effect can be smaller than the introduced compositional overhead. Furthermore, we have shown that dynamic binding may also introduce

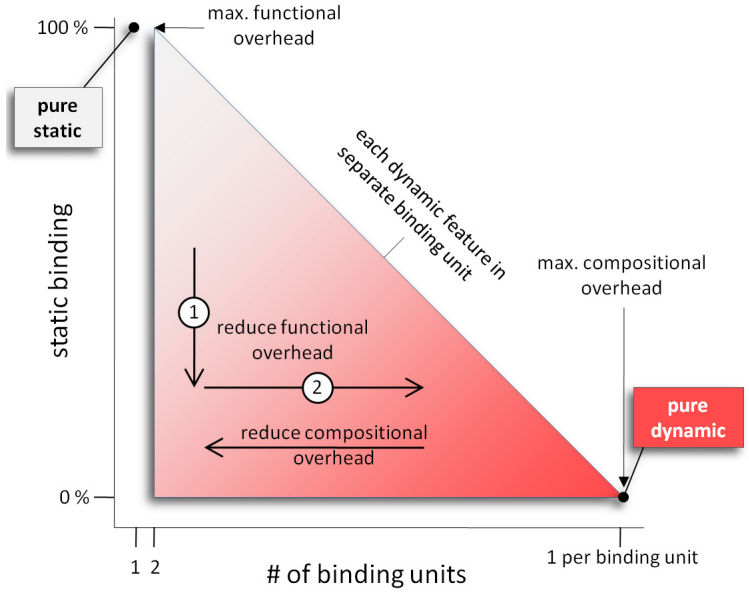


Figure 6.16: Combining static and dynamic binding to support a varying number and varying size of binding units. Possible configurations of dynamic binding units are shown as a triangle.

an overhead due to unused methods that can be removed by the linker when using static binding. An advantage of our approach is that it allows a programmer to find a balance between compositional and functional overhead that is suitable for her needs.

As illustrated in Figure 6.16, our approach provides purely static and purely dynamic binding (upper left and lower right corners) as well as all combinations with varying sets of binding units (shown as triangle). When creating binding units, the compositional overhead can be reduced for a constant number of dynamically bound features by increasing the number of features per binding unit (arrow in lower part of Figure 6.16). The functional overhead can be reduced in two ways: On the one hand, increasing the number of dynamically bound features for a constant number of binding units (i.e., moving features from the base program into a binding unit) reduces the size of the base program (arrow (1) in Figure 6.16). On the other hand, increasing the number of dynamic binding units for a constant number of dynamically bound features (i.e., splitting the binding units) reduces the size of each binding unit (arrow (2) in Figure 6.16).

6.5.2 Customizability and SPL Development

Generating binding units from FOP code has an effect on customizability of an SPL and improves several aspects of SPL development.

Granularity and Flexibility. The dynamic binding capabilities of FeatureC++ enable developers to achieve extensibility of a program after deployment. Additionally, features can be bound statically, which supports fine-grained extensions without increasing the execution time. For example, the B-Tree in FAME-DBMS is built from many small features (e.g., features for write support) that can be statically configured. For binding units, we thus achieve performance and memory consumption comparable to static binding with the C preprocessor (cf. Chapter 4). Fine-grained extensions and dynamic binding are opposite optimization goals with respect to performance and memory consumption. The more fine-grained the extensions are, the more memory and computing power is required for dynamic binding (cf. Figure 6.15). Our approach allows a programmer to combine both binding times as needed per application scenario.

Reuse. The combination of both binding times increases reuse possibilities in different application scenarios. For example, we can statically bind all features of FAME-DBMS for deeply embedded devices and support dynamic binding for other platforms. Furthermore, binding time flexibility enables reuse of features across different SPLs that use different binding times. For example, we can reuse a feature that implements a communication protocol in an e-mail client SPL that uses dynamic binding and also in an e-mail server that uses static binding.

Crosscutting Features. With our approach, static binding can also be used for crosscutting features that are spread across multiple dynamic binding units. These features are usually implemented with preprocessors [Gri00] or design patterns [MSL00, Zdu04]. Adding or removing such features is possible by rebuilding the affected binding units. For example, in FAME-DBMS, WRITESUPPORT is a crosscutting feature that affects several binding units such as the query engine, indexes, etc. Using FeatureC++, we can add or remove this feature and have to rebuild only the affected binding units.

Development and Maintenance. Using a single mechanism for implementing features (i.e., feature modules) also simplifies SPL development. A programmer may combine FeatureC++ with other variability mechanisms (design patterns, macros, `#ifdefs`, etc.) or may replace other mechanisms by using feature modules exclusively. Binding time flexibility can simplify maintenance of an SPL. For example, dynamic binding can be replaced temporarily by static binding for debugging purposes to avoid the complexity of dynamic binding. Finally, the presented approach provides means for step-wise transition from static to dynamic binding and thus reduces the adoption barrier for dynamic binding.

6.5.3 A Guideline for Defining Binding Units

When configuring an SPL for static and dynamic binding, we have to answer two questions: Which features have to be bound dynamically? Which dynamically

bound features should be composed into the same binding unit? With our approach, a domain expert can decide this per application scenario. Static binding does not exhibit any compositional overhead. It is usually the best choice if extensibility after deployment or at runtime is not required. The remaining challenge for a domain expert is to find proper binding units for dynamically bound features to provide the required flexibility while minimizing the overhead. Therefore, resource consumption of different feature combinations has to be analyzed, which means a high effort and may be impractical. The following rules can be used to find good feature combinations for binding units more easily:

1. As a simple rule, a large number as well as a large size of binding units should be avoided because the first increases the compositional overhead and the latter increases the functional overhead. However, as depicted in Figure 6.16, this cannot be a fixed rule because reducing one overhead may increase the other.

2. Analyzing the feature model helps to find features that should be combined in a binding unit. For example, atomic feature sets (i.e., features that cannot be configured independently [ZZM04]) have to be part of the same binding unit. Similarly, a *requires*-constraint between two features indicates that the features are used in combination and may also be combined into the same binding unit.

3. Furthermore, we can analyze the source code of features. A high degree of coupling between features indicates which features are commonly used together [AB11]. Hence, it can be beneficial to merge them into a single binding unit. Crosscutting features should be bound statically if possible. An automated analysis of coupling and cohesion could be used to provide an initial assignment of dynamically bound features to binding units.

4. Implementation knowledge can be used to find features and methods that are important with respect to performance and memory consumption. For example, frequently called *hot spot* methods should ideally be bound statically. If this is not possible, they should be defined in a single binding unit only. This causes the method to be bound dynamically but avoids a decomposition of the method into multiple fragments. Similarly, when allocating a large number of small objects (such as list elements), the corresponding class should be defined in a single binding unit.

To further reduce the overhead of a program, different optimizations of binding units are possible. For example, *overlapping binding units* (i.e., binding units that use an overlapping set of features) can be created to provide binding units with a small interface or to reduce the number of binding units. Another optimization is to *split* or *merge* binding units when the requirements have changed over time or when an analysis at runtime has identified how the binding units are actually used. For example, binding units that are often or always used in combination can be merged into a single binding unit without changing the source code.

6.6 Related Work

We aim at combining static and dynamic binding of features using a generative approach that allows us to use the same code base for different binding times. There are several collaboration-based and feature-based approaches that support either static or dynamic binding as described in Chapter 3. In contrast to these approaches, our solution supports both, static and dynamic composition, and assists the developer in dynamically composing SPLs and validation of SPL configurations according to a feature model before SPL instantiation. In the following, we compare our solution to other approaches that support static and dynamic binding.

Support for Different Binding Times. There are approaches for software composition that employ different techniques or paradigms to support different binding times. For example, CaesarJ [AGMO06] supports static composition based on virtual classes and dynamic deployment of aspects. Object Teams [Her02] support dynamic composition of *teams*, which can be used to implement features. The approach also supports static composition by declaring a team as `static`. This has to be done in the source code at development time. Components and the C preprocessor are also combined to support different binding times in practice. Both approaches require to know the binding time of an implementation unit at design time. By contrast, we can choose the binding time at deployment time to enable reuse of source code even when using different binding times.

JavAdaptor allows to reconfigure a running program and can be combined with Jak [PSC09]. This can be used to support binding of whole features at runtime and supports static optimization of the whole program before applying the changes. Reconfiguration with JavAdaptor means to compile a new program and to apply the changes, which is time consuming and thus inappropriate for frequent updates. Furthermore, the approach does not allow to use multiple instances of an SPL at the same time or to manipulate an SPL instance from client code.

Zdun et al. introduce transitive mixins to generalize composition of classes and objects [ZSN07]. The implementation provided in [ZSN07] is built on top of a dynamic approach that does not support static composition. Chakravarthy et al. provide with Edicts an approach that supports different binding times using design patterns that are applied to a program by means of aspects [CRE08]. Configuration is done by switching between Edicts. Schmid et al. support different binding times for the same code fragment by combining AspectJ with conditional compilation using if-statements in Java [SE08b]. Czarnecki et al. describe how to parameterize the binding time using C++ templates [CE00]. They provide a configurable binding time (e.g., for class extensions) with a template-based program generator. The OSGi[6] standard also supports static and dynamic composition of components, called *bundles*. It is a component-based approach that does not aim at implementing (potentially crosscutting) features. It thus does not provide means to flexibly generate

[6] http://www.osgi.org

larger bundles on demand by statically merging a set of features when they are implemented as bundles. Other approaches support static and dynamic binding of aspects. AspectC++ supports weaving at runtime and compile-time for the same aspect [GS05]. AspectJ supports weaving advice at compile-time, after compile-time, and at load-time.[7] PROSE [NAR08], Steamloom [BHMO04], Hotwave [VBAM09], and other AspectJ-based approaches support weaving at runtime and may be combined with AspectJ's static weaving. These AOP approaches can be used to support multiple class extensions at the same time as in FOP.

In contrast to FeatureC++, the approaches above do not provide a mechanism for feature composition according to a feature model. Nevertheless, the approaches can be combined with tools that support configuration and static composition of SPLs such as pure::variants [pur04]. However, this is not possible for validating configurations at runtime. By using FOP and an SPL-aware type system, we can ensure type-safety for a whole SPL. Using a static feature model transformation, we can even ensure type safety for dynamic composition of binding units.

Similar to our approach, the mechanisms above combine static and dynamic binding. Approaches such as Edicts and AspectC++ can be used to bind a feature statically or dynamically with the base program without changing the implementation. However, they do not provide means to statically merge an arbitrary set of dynamically bound features into a single binding unit and compose the binding units dynamically. One reason is that these approaches do not preserve the execution order of method extensions when mixing both binding times in this way (e.g., aspect ordering in AspectC++ [TLSS10]). Applying static binding first and dynamic binding afterwards changes the feature composition order (all static features are bound before dynamic features). This in turn may change the behavior of methods that are refined by statically *and* dynamically bound features (e.g., statically and dynamically weaved aspects that have the same join point). We solve this by generating hook methods, as described in Section 6.3.

Binding Units. Lee et al. suggest to decide before SPL development which features to implement in one component and to combine the resulting components at runtime [LK06]. Griss argues that components and novel approaches to software composition should be combined to develop SPLs [Gri00]. He discusses different approaches that may be used to customize components when the feature selection changes. Our approach goes into the same direction. However, we use only a single implementation technique that supports static and dynamic binding. We also think that components have to be planned before SPL development, but the selection of concrete features and component customization has to happen at deployment time.

Staged Configuration. Czarnecki et al. [CHE05] and Classen et al. [CHH09] define staged configuration as a process that has to eliminate configuration choices. We do not adhere to this strict definition and define a configuration step as a set

[7] http://eclipse.org/aspectj

of constraints over the features of an SPL. This was also described by Classen et al. [CHH09] and it can be easily checked whether a configuration step reduces the variability by using a SAT solver [TBK09]. In contrast to the specialization steps defined by Czarnecki et al. [CHE05], we support arbitrary configuration constraints. For generating dynamic binding units, we improve visualization of specialized feature models using a refactoring similar to the refactorings for staged configuration described by Czarnecki et al. For these refactorings, we use the atomic steps described by Alves et al. [AGM+06].

Our approach for stepwise program composition can be compared to model-driven software development (MDD) [SV06], which aims at step-wise specialization of models. In MDD, variability is also encoded in the transformation system. For example, platform-specific variability is usually realized via transformations of a platform-independent model into a platform-specific model [DSGB03]. We describe parts of the variability of the transformation system (in our case FeatureAce) using a feature model. Users can thus tailor the product derivation process in the same way as they configure the functional features of the SPL. This is similar to the approach by Heidenreich et al., who use FeatureMapper [HKW08] to map features to model transformations to achieve safety of transformations in MDD [HKA10].

6.7 Summary

With the presented approach, we unify static and dynamic feature binding, which are currently used for software development in several domains. Our approach is based on FOP and allows a programmer to develop all features of an SPL with the same implementation technique, namely *feature modules*. After development, the used binding time can be chosen per feature. This flexibility allows an application engineer to choose the binding time depending on the requirements of each application scenario. Furthermore, it enables reuse of features between SPLs that require different binding times.

In contrast to other approaches that support static and dynamic binding of the same code unit, we provide an extended solution:

- We support *feature-based configuration*, which means to compose *whole feature modules*. This is required because (1) a feature usually contains several classes and class fragments that have to be composed as a whole and (2) we achieve composition safety by using an SPL's feature model to validate configurations before dynamic composition.

- We generate *dynamic binding units* to statically compose all features that are bound at the same time. This optimizes resource consumption even for dynamically bound features.

By extending staged configuration, we provide the foundation for validating static and dynamic composition steps. The presented extensions are independent of the concrete binding time and support a general process of *staged composition* of feature

modules. By using a transformed feature model that corresponds to the composed binding units we achieve safe composition independent of the binding time and the used binding units. With a type system for SPLs, we could furthermore achieve type safety. This is also the basis for feature-based runtime adaptation of binding units, which we present in the next chapter.

We provide a prototypical implementation of the approach on top of FeatureC++. By extending the FeatureC++ compiler we were able to apply the approach to non-trivial case studies. In an evaluation, we could show that generated binding units provide both, flexibility due to dynamic binding and resource optimization due to static composition. We could show that the approach can be used instead of a mixture of different approaches as it is currently done in practice. For example, it can be used as an implementation mechanism for plugins while replacing the C preprocessor for static binding of other features.

Finally, the approach scales from purely static to purely dynamic composition. When using purely static composition, the approach avoids any compositional overhead and generates code that is optimized with respect to resource consumption. It thus provides the same performance as the C preprocessor. On the other end of the spectrum, we provide highest flexibility due to dynamic binding. In between, our approach enables application engineers to find the optimal balance between the different binding times.

7 Runtime Adaptation with Dynamic Binding Units

This chapter shares material with the GPCE'11 paper "Tailoring Dynamic Software Product Lines" [RSPA11].

The approach presented so far and SPLs in general, usually support static binding or dynamic binding at load-time (e.g., by composing components). By contrast, *dynamic software product lines (DSPLs)* aim at variability at *runtime* to cope with changing requirements during program execution. DSPLs often implement variability with components and use the high-level software architecture to describe program adaptations [OGT+99]. They allow a programmer to specify adaptation rules for reconfiguring components and thereby abstract from the concrete implementation [GCH+04, FHS+06, HZP+07]. Consequently, an adaptive system can also be considered as a DSPL [BSBG08]. Besides approaches that use architectural models to describe program adaptations, there are also DSPL approaches that support feature-based runtime adaptation. For example, some approaches use *feature models* to describe dependencies between features and to reason about runtime variability of DSPLs and adaptive systems [CGFP09, HSSF06, LK06, TRCP07]. Describing also program adaptations in terms of features abstracts from implementation details, simplifies reconfiguration of running programs, and allows for checking consistency of adaptations [CGFP09]. Such feature-based approaches use a mapping of DSPL features to components that are used for implementation [LK06, TRCP07]. However, components are usually coarse-grained to minimize complexity and the compositional overhead at runtime. This limits customizability and applicability of a DSPL.

In this chapter, we present an approach for feature-based runtime adaptation and self-configuration that is based on tailor-made binding units. We generate a DSPL from a feature-oriented implementation of an SPL by generating binding units (cf. Chapter 6) and integrating generated code for runtime adaptation into the DSPL. We achieve this by extending FeatureAce (cf. Sec. 5.3.2) with adaptation and self-configuration capabilities. In a generated DSPL, multiple features of the original SPL map to a compound feature of the DSPL (i.e., to a dynamic binding unit). While we use the compound features for adaptation at runtime, we are able to describe and validate program adaptations in terms of the SPL features that are independent of the actually used binding units. This allows us to bridge the gap between feature-based variability modeling and runtime adaptation with coarse-

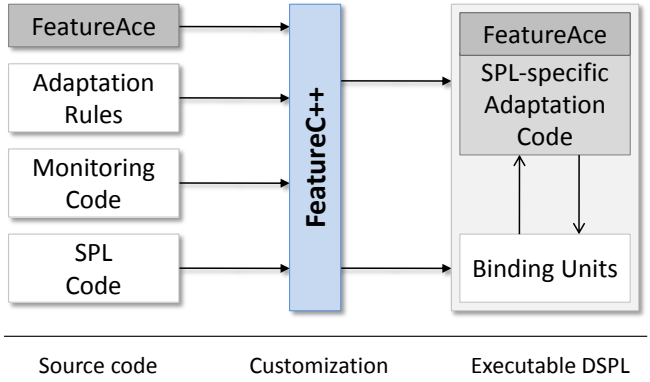


Figure 7.1: Generating a DSPL from FeatureAce, user-defined extensions of FeatureAce, and an SPL's implementation.

grained modules. With the presented approach, we are able to generate DSPLs with minimal resource requirements. We furthermore reduce the effort for calculating a configuration of a DSPL by minimizing the number of dynamically bound modules. To demonstrate applicability of our approach we evaluate it with a case study.

7.1 Feature-based Runtime Adaptation

We support feature-based runtime adaptation in DSPLs by describing configuration changes in terms of features and by applying these changes to the feature model before composing the corresponding binding units. In the following, we call the compound features, which represent binding units, the *features of the DSPL* and use a feature model to describe their dependencies. Since the features of an SPL often represent requirements, there is a direct mapping of changing requirements to changes of an SPL's configuration. Due to the use of binding units, there is an n-to-1 mapping of SPL features to DSPL features. For simplifying the description of the adaptation mechanisms, we assume a 1-to-1 mapping of SPL features to DSPL features in the following. However, as we describe in the next section, we support an n-to-1 mapping by transforming the SPL feature model (cf. Sec. 6.2.3) and the adaptation rules according to the generated binding units. To support autonomous configuration of DSPLs, we extended FeatureAce with runtime adaptation and self-configuration capabilities. Next, we introduce the FeatureAce extensions required for runtime adaptation.

7.1.1 A Customizable Adaptation Framework

The FeatureC++ compiler generates a DSPL from an SPL's implementation, monitoring code, adaptation rules, and the code of FeatureAce, as illustrated in Figure 7.1. In the resulting executable DSPL, a metaprogram is responsible for au-

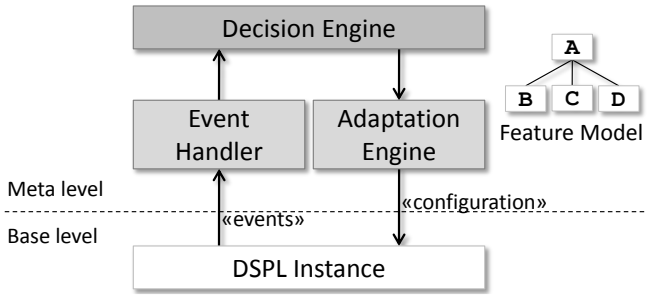


Figure 7.2: Architecture of a DSPL: Domain code is at the base-level and adaptation code is at the meta-level.

tonomous composition and self-adaptation at runtime. The generic metaprogram has access to the base-level (i.e., the binding units) via generated SPL-specific adaptation code.

As shown in Figure 7.2, FeatureAce provides a decision engine that uses the feature model of the DSPL to reason about validity of changes in the running program. Monitoring code for analyzing the context at runtime is located at the base-level. It is developed as part of the SPL since it is usually highly domain specific. For example, code for monitoring DBMS queries can be used to trigger an event for loading a feature that implements a special search index when a particular kind of query is detected. The events triggered by the monitoring code are captured by an event handler that activates the decision engine. Based on adaptation rules, the decision engine calculates a new configuration. The adaptation engine applies configuration changes by loading and unloading binding units.

To seamlessly integrate static and dynamic binding, we provide means to statically customize the adaptation infrastructure:

- Monitoring code of the DSPL that triggers adaptation events is implemented in distinct features. Hence, it is possible to use only required monitoring code and to choose between alternative implementations.
- Adaptation rules are also stored in separate feature modules to allow the programmer to choose only required adaptation rules at deployment time.
- Since we developed FeatureAce as an SPL, a user can customize it to choose between manual and autonomous adaptation and to enable validation of adaptations only if required.

Customization of the adaptation infrastructure allows us to cope with changing requirements with respect to the adaptation process (e.g., due to changes in the execution environment). In the following, we use static binding for customization of the adaptation infrastructure. In general, parts of this variability may also be required at runtime, which requires dynamic binding of selected features of FeatureAce and adaptation code. This is beyond the scope of this thesis but it illustrates that there are further challanges for improving the flexibility of feature composition.

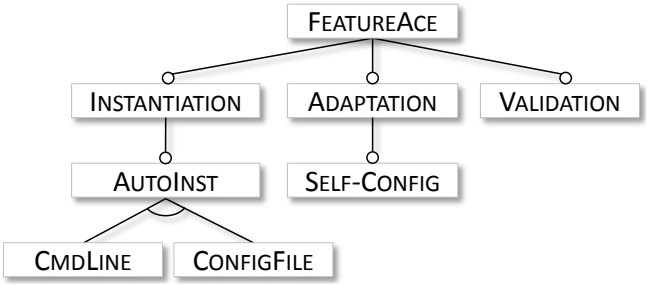


Figure 7.3: The feature model of FeatureAce. Customization of dynamic product instantiation and runtime adaptation capabilities is achieved by selecting the corresponding features.

In Figure 7.3, we show the feature diagram of FeatureAce from Chapter 5. Feature AutoInst encapsulates the functionality required for automated SPL instantiation using command line arguments or a configuration file to provide an initial feature selection. Feature Adaptation enables modification of a running SPL instance and feature Self-Config supports rule-based self-configuration. Feature Validation provides functionality to check validity of an SPL variant before composing the binding units. For customization, a user defines the required adaptation facilities of FeatureAce or may add user-defined extensions. FeatureAce extensions (e.g., monitoring code) are implemented as additional feature modules without the need for invasive modifications of FeatureAce. According to the feature selection, the code of FeatureAce extensions and adaptation rules is also statically composed. Hence, only selected adaptation rules and adaptation code are included in a DSPL.

7.1.2 DSPL Adaptation

FeatureAce supports a set of operations for instantiation and adaptation of a DSPL at runtime:

DSPL Instantiation: A DSPL instance is composed from multiple *feature instances*. The result is a stack of feature instances that represents a DSPL instance (cf. Chapter 5).

DSPL Adaptation: An already running DSPL instance can be modified by adding and removing features as well as activating and deactivating already loaded features.

Note that not every feature that can be bound at load-time can also be bound at runtime without further modification. For example, runtime adaptation requires to support consistent changes with respect to the state of features (e.g., initialization of objects). Data for initialization that is available at load-time is not always available at runtime, which may render runtime adaptation impossible. There are further

issues such as concurrency control and state transfer that have to be considered for transition from an SPL to a DSPL [SE08a]. As a solution, developers may provide special code for binding at runtime. Using FOP, this code can be separated in feature modules that are only included in a program when runtime adaptation is used. However, such special requirements on runtime adaptation are beyond the scope of this thesis and we only provide the foundation for generating tailor-made DSPLs.

We have already described the instantiation capabilities of FeatureAce in Chapter 5. In the following overview, we describe the supported adaptation mechanisms:

Add and remove features: A feature instance can be applied to or removed from a running SPL instance. A feature that is not part of any running SPL instance can be deleted and unloaded.

Activate and deactivate features: A feature instance can be deactivated if it is temporarily not needed and can be reactivated later. This maintains the state of a feature while disabling its functionality.

Exchange features: Compatible feature instances (features that provide the same data layout) can be exchanged in a single operation to reuse the state of a running feature module in a new feature module. In contrast to removing the feature and adding a new feature, this operation works without changing the data and is thus much faster. This way, alternative algorithms can be exchanged with a minimal runtime overhead.

The described operations are internally used by FeatureAce for runtime adaptation but can also be accessed from an external program or from the base-level. We support access for external programs via the network interface of FeatureAce. The base-level can access the adaptation API of FeatureAce via a reflection mechanism.

Self-Adaptation via Reflection

We provide a *reflection* mechanism that allows the base-level of a DSPL to access the adaptation meta-level for observing or modifying the current configuration. For example, objects can access their DSPL instance for querying information and also for modifying the configuration. The reflection mechanism is sufficient for adaptations that do not require complex adaptation rules, i.e., when events can be directly mapped to a configuration change. For example, a programmer can write adaptation code to activate a feature when the user selects a particular menu entry in the running program. In Figure 7.4, we depict an example for activating an already loaded feature (a feature for database access in a query analyzer client) by accessing the meta-level. This mechanism simplifies application development since no additional code for a metaprogram for runtime modifications has to be written. However, the adaptation code at the base-level entangles runtime adaptation and DSPL implementation, which may complicate evolution of the DSPL. For example, renaming a feature requires to rename it also in the implementation.

```
1 void OnDbActivate(string dbType) {
2   //activate the selected db type
3   FeatureConfig::ActivateFeature(dbType);
4 };
```

Figure 7.4: FeatureC++ source code for activating a feature from the base-level.

Hence, it is often more appropriate to separate domain implementation and meta-level. This can be done by separating adaptation code in distinct feature modules. A more flexible mechanism that is independent of the implementation of an SPL is to use adaptation rules, as we describe in the next section.

Runtime Validation of Feature Compositions

FeatureAce uses the feature model of the DSPL to validate a feature selection at runtime. To provide only the required variability at runtime, we transform the original SPL feature model according to the generated DSPL, as described in Section 6.2.3. FeatureAce uses the transformed feature model for calculating a new configuration at runtime and for validation of configuration changes. Combining this with static type checking of the SPL, we achieve type-safe adaptation at runtime.

7.2 Rule-based Adaptation

FeatureAce provides a rule-based mechanism for self-adaptation that allows us to separate application logic from runtime adaptation mechanisms. In contrast to most existing approaches, an adaptation rule is described in terms of features, which requires a special mechanism for computing a configuration using *configuration constraints*.

7.2.1 Configuration Constraints

Our approach for runtime adaptation is based on adaptation rules that describe how a configuration of an SPL must be changed when an event is triggered. A configuration C of a program P of a DSPL is the set of features that is included in P. During adaptation, we derive a configuration C from a set of requirements R that define which features of the DSPL must be included in a valid program. In the simplest case, the requirements are a set of required features (e.g., a user-defined feature selection). In general, however, a requirement may be an arbitrary *configuration constraint* (i.e., a propositional formula over the set of available features) that restricts the set of valid configurations [RSTS11]. A configuration constraint is not different from a domain constraint of a feature model but it is added and removed during configuration. For example, to express that a feature must be included in a program we can define a *requires*-constraint for that feature.

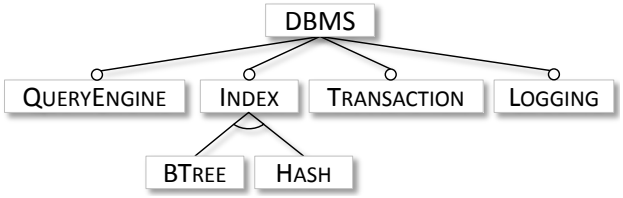


Figure 7.5: Feature model of a simple DBMS.

As an example consider the feature diagram of Figure 7.5 with an initial set of requirements R (e.g., defined by a user):

$$R = \{\text{QUERYENGINE}, \text{INDEX}\} \tag{7.1}$$

R defines that features QUERYENGINE and INDEX must be included in a valid configuration. We can thus derive a configuration C that satisfies R:

$$C = \{\text{QUERYENGINE}, \text{INDEX}, \text{HASH}\}. \tag{7.2}$$

Because C must also satisfy the constraints defined in the feature model, such as the XOR-constraint between HASH and BTREE (cf. Fig. 7.5), it must include one of the two features. In our example, we have chosen feature HASH. While C represents a single program, R defines a *specialization* S of our DSPL that represents multiple configurations, as illustrated in Figure 7.6. DSPL D has two specializations S_1 and S_2, which we denote with inheritance [RSTS11]. Each specialization represents multiple configurations (illustrated with a cone). For example, C_1 and C_2 are configurations of S_1. Each specialization represents a subset of the configurations of the unspecialized DSPL.

We can represent a set of requirements R as a single propositional formula using a conjunction of all requirements. For example, R from equation (7.1) corresponds to the boolean constraint QUERYENGINE $\wedge$ INDEX. Since a feature model can also be translated into a propositional formula [Bat05], we can check if a configuration C satisfies the requirements R for a feature model FM: If $FM \wedge R$ is `true` for configuration C then C is valid with respect to R. Furthermore, we can use a SAT solver to test if R is a valid set of requirements with respect to FM. This can be done by checking if we can derive at least a single valid configuration, i.e., $FM \wedge R$ must be satisfiable [TBK09]. This allows us to check at runtime whether a specialization S_i has a valid configuration (cf. Figure 7.6).

7.2.2 Adaptation Rules

The current configuration C of a running DSPL is modified by a configuration change ΔC (i.e., a reconfiguration) that defines which features are added to C and which features are removed from C during adaptation. However, as we explain below, it is usually too restrictive to directly define configuration changes in an adaptation rule.

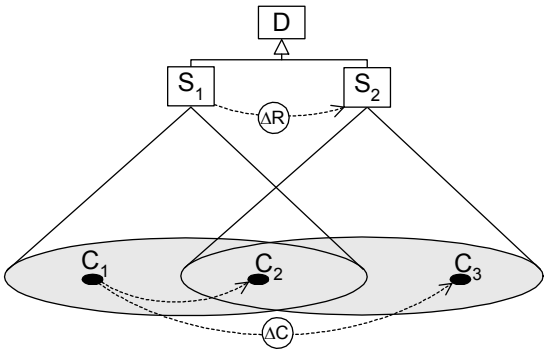


Figure 7.6: DSPL D with specializations S_1 and S_2 and configurations C_1-C_3.

Instead, an adaptation rule describes changes with respect to the active requirements R of a DSPL. We thus define an adaptation rule A as a pair $(E, \Delta R)$ where E is the event that triggers rule A and ΔR are modifications that must be applied to R when E is triggered. ΔR is a pair $(\Delta R_\oplus, \Delta R_\ominus)$ of added and removed requirements. We use operator $\bullet$ to denote adaptations (i.e., application of ΔR to R):

$$R' := \Delta R \bullet R \tag{7.3}$$
$$:= (R \setminus \Delta R_\ominus) \cup \Delta R_\oplus. \tag{7.4}$$

R' must be satisfied after applying rule A to a configuration C. That is, ΔR does *not* directly modify the configuration of a DSPL but it modifies a set of requirements R that describe a specialization of the DSPL. As illustrated in Figure 7.6, applying ΔR to S_1 results in a different specialization S_2.

From a modified set of requirements R', we derive a modified configuration C'. In Figure 7.6, we can derive two valid configurations C_2 or C_3 from S_2. For runtime adaptation, we have to choose one of these configurations. For example, we may choose a configuration with the smallest number of features. Finally, we derive the corresponding configuration change ΔC, which is a pair $(\Delta C_\oplus, \Delta C_\ominus)$. It defines the set of features that must be added ($\Delta C_\oplus$) and removed ($\Delta C_\ominus$) for adaptation. We compute it from the current configuration C and the target configuration C':

$$\Delta C := (\Delta C_\oplus, \Delta C_\ominus) \tag{7.5}$$
$$\Delta C_\oplus := C' \setminus C \tag{7.6}$$
$$\Delta C_\ominus := C \setminus C' \tag{7.7}$$

$\Delta C_\oplus$ is the set of features that are added to C and $\Delta C_\ominus$ are removed from C during adaptation. As a complete example, consider the DBMS from equations (7.1) and (7.2) with an adaptation rule A that is triggered on event E_{Range}, meaning that range queries are frequently used:

$$A = (E_{Range}, (\{\textsc{Btree}\}, \emptyset)) \tag{7.8}$$

```
1 //Load transaction management
2 BeginTxn : OnTxn => addConstraint(TX: Transaction)
3
4 //Process range queries
5 BeginRQ : OnRangeQuery => addConstraint(RQ: Btree)
6
7 //Remove constraint RQ
8 EndRQ : OnRangeQueryEnd => removeConstraint(RQ)
```

Figure 7.7: Two adaptation rules that add named constraints TX and RQ (Lines 2 and 5) and a rule that removes constraint RQ (Line 8).

$$
\begin{aligned}
R' &= (\{\text{BTREE}\}, \emptyset)) \bullet R && (7.9)\\
&= \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} && (7.10)\\
C' &= \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} && (7.11)\\
\Delta C &= (\{\text{BTREE}\}, \{\text{HASH}\}). && (7.12)
\end{aligned}
$$

Rule A adds feature BTREE to R (i.e., BTREE has to occur in a valid configuration), which results in the modified requirement R'. From R' we derive a new configuration C'. The required configuration change ΔC (adding feature BTREE and removing feature HASH) is derived from C and C' according to equation (7.5).

In contrast to modifying the set of requirements, a direct configuration change is too restrictive. For example, an adaptation rule that adds feature BTREE directly to configuration C violates the XOR constraint of the feature model (either BTREE or HASH must be selected). Furthermore, direct configuration changes do not allow us to define rules that remove features because a removed feature may be required by other constraints.

Specifying Adaptation Rules. We specify adaptation rules in a declarative language, as shown in the example in Figure 7.7. The corresponding grammar is shown in Figure 7.8. A rule consists of a name, a named adaptation event E (e.g., `OnTxn` in Line 2) that triggers execution of a set of actions ΔR, which modify the current configuration of a DSPL. An action can add or remove a named configuration constraint using keywords `addConstraint` and `removeConstraint` followed by a constraint definition (Line 5) or a constraint name respectively (Line 8). Each constraint has a name to be able to remove it from the requirements of a DSPL, as shown in Line 8.

Currently, we define adaptation events in monitoring code using the host language. It would also be possible to use a declarative specification as it is done in related approaches [GCH+04].

```
1  AdaptScript: Rule+ ;
2  Rule: RuleName ":" EventName "=>" Action+ ";" ;
3  RuleName: ID ;
4  EventName: ID ;
5  Action: AddReq | RemoveReq ;
6  AddReq: "addReq" "(" ReqName ":" Constraint ")" ;
7  RemoveReq: "removeReq" "(" ReqName ")" ;
8  ReqName: ID ;
9  Constraint: FeatureName | "!" Constraint | "(" Constraint ")"
10           | Constraint ConstraintOp Constraint ;
11 ConstraintOp: "&&" | "||" | "->" | "<->" ;
12 FeatureName: ID ;
```

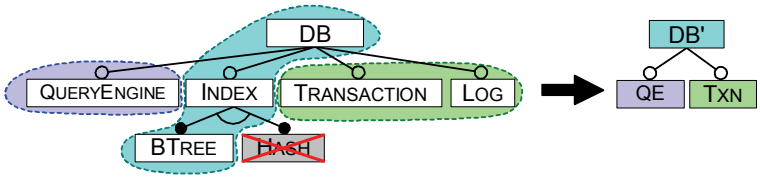
Figure 7.8: Grammar of FeatureAce's adaptation rule specification language.

Applying Adaptations

Before computing a new configuration when applying an adaptation rule, FeatureAce checks whether an adaptation is really needed: If a set of requirements R_i represent a specialized DSPL S_i (e.g., S_1 in Figure 7.6) then $R_{i+1} = \Delta R \bullet R_i$ corresponds to a new specialization S_{i+1} (e.g., S_2 in Figure 7.6). If S_i and S_{i+1} are overlapping then there are configurations that can be derived from both specializations (e.g., C_2 in Figure 7.6). Hence, if the current configuration of the DSPL is also a valid configuration of the new specialization, we do not have to adapt the running program. For example, adaptation of S_1 to S_2 in Figure 7.6 for configuration C_2 does not require a program adaptation. Hence, the decision engine of FeatureAce first checks whether the current configuration C_i satisfies the new requirements R_{i+1}.

If this is not the case, we have to find a new configuration C_{i+1} that satisfies R_{i+1}. To test if there is any valid configuration that satisfies R_{i+1}, the decision engine checks satisfiability of the feature model including the changed requirements. If there are multiple valid configurations, the decision algorithm has to choose the *best* one. Which configuration is the *best* depends on several requirements and is a challenging task in current research [FHS+06, WDS09]. For example, we may choose the configuration with the smallest number of features or the smallest number of required adaptations. Other optimization goals are non-functional requirements such as memory consumption, performance, or quality of service. We currently choose the configuration with the smallest number of configuration changes to reduce the number of required adaptations. For that reason, FeatureAce tries to keep already configured features to minimize changes. Features are removed from a configuration when they violate a constraint. Hence, when an adaptation rule removes a constraint from the requirements R, this does not always cause a configuration change. Furthermore, we remove features that are not required for a configurable time span to provide a simple mechanism for reducing resource consumption due to features that are not used.

To reduce consumed resources, a configuration can also be explicitly minimized




```
addConstraint(TX: Transaction)
addConstraint(RQ: Btree)
removeConstraint(RQ)
```

```
addConstraint(TX: Txn)
addConstraint(RQ: DB')
removeConstraint(RQ)
```

Figure 7.9: Transformation of a feature model (cf. Fig. 6.4) and the corresponding transformation of actions of adaptation rules (lower part) according to the defined binding units. Transformations in adaptation rules are underlined.

by rules, e.g., triggered by low working memory. In future work, we plan to use more sophisticated mechanisms to trigger unloading of features based on non-functional requirements. For example, we may remove unused features based on statistics and the workload of the system or we may optimize a configuration using CSP[1] solvers [BRCT05, WDS09].

Adaptation Rules for Binding Units

Above we assumed that there is a 1-to-1 mapping of SPL features to the features of the DSPL (i.e., the binding units). In practice, however, there is an n-to-1 mapping of SPL features to DSPL features because multiple features of the SPL are merged into a single binding unit of the DSPL. As described in Section 6.2.3, we transform the SPL's feature model according to the chosen binding units and thus derive the feature model of the generated DSPL. In the upper part of Figure 7.9, we show an example of a feature model transformation when generating a DSPL. According to this transformation, we apply a corresponding transformation to the adaptation rules that have been defined with respect to the original SPL features (lower part of Figure 7.9). This transformation is easy to achieve by replacing each feature in adaptation rules with the binding unit of the feature in the generated DSPL. For example, we have to replace all occurrences of feature TRANSACTION with its corresponding binding unit TXN. After transformation, requirement `RQ` is always valid (second action in Figure 7.9) because `DB'` is the root of the feature tree and included in every configuration. Hence, we can remove all actions that add or remove requirement `RQ` such as the last two actions in Figure 7.9. This does not affect rules which require that feature BTREE must not be included in a concrete variant (e.g., using an action that adds a requirement ¬`Btree`), which results in an invalid rule as discussed next.

After transformation, we can check if all adaptation rules can be applied to the DSPL feature model by using a SAT solver. For example, an adaptation rule that

[1] Constraint satisfaction problem

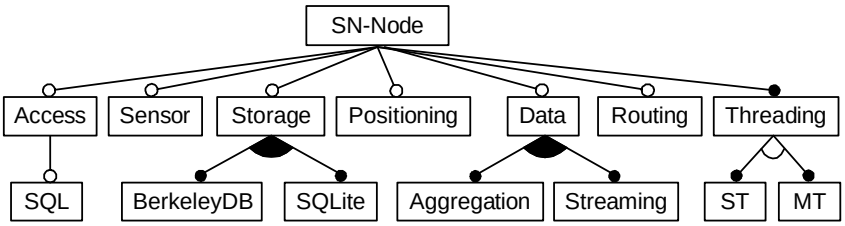


Figure 7.10: Feature diagram of an SPL for sensor network nodes.

adds requirement ¬`Btree` is transformed into a rule with requirement ¬`DB'`, which is not satisfiably since *DB'* is the root of the feature model and is thus always true. In such a case, the rule is invalid with respect to the generated DSPL (i.e., with respect to the chosen binding units). As another example consider features Transaction and Log in Figure 7.9. Both features are part of the binding unit Txn. Hence, both features are always present at the same time in the generated DSPL. Consequently, an adaptation rule that requires *either* Transaction *or* Log cannot be used with this DSPL. Using a SAT solver, we can detect such invalid rules already before runtime.

7.3 Case Study and Discussion

By means of a case study, we demonstrate the flexibility of our approach and show that binding units can reduce the time needed for runtime adaptation. We use a prototypical implementation of FeatureAce for FeatureC++. However, the concept can be applied to other languages as well. As application scenario, we use a sensor network simulation.

7.3.1 An SPL for Sensor Network Nodes

A sensor network (SN) is a network of interconnected embedded devices (e.g., via radio communication), which sense different kinds of information (temperature, light, etc.) [MI06]. There are different types of nodes in a sensor network. *Sensor nodes* measure data, store it locally, and send it to other nodes. *Aggregation nodes* aggregate data (e.g., computing the mean value) from other nodes. *Access nodes* provide access to clients that connect to the network from the outside.

By using an SPL of node software, we can *generate* different program variants tailored to the different kinds of SN nodes. In Figure 7.10, we depict an excerpt of the feature diagram of an SN-Node product line we implemented in FeatureC++. Subfeatures of Data are used for aggregation in data nodes (Aggregation) and streaming in access nodes (Streaming). A node can not only play a single role (e.g., being a sensor node) but multiple roles at the same time. For example, a node may aggregate data but may also be responsible for accessing the network. To compensate node failures and for efficiency, the role may change over the lifetime

Hardware	Role	Binding Units
Simple	Sensor	StaticSense
Advanced	Positioning	Core, Positioning
	Sensor	Core, Sense
	DataAggregator	Core, QueryProc, Aggregation
	AccessNode	Core, QueryProc, Streaming

Table 7.1: Examples of different roles and their binding units for two kinds of devices.

of a node. For example, if the access node fails due to exhausted battery power, a different node can reconfigure itself to provide this service. Due to hardware constraints, not all physical nodes can play every role. For example, only a node with sufficient storage capacity can be used for data aggregation. Such limitations influence the configuration process when defining binding units at deployment time:

1. *Dynamic Binding:* For embedded devices that do not allow dynamic changes to loaded program code (because the executable code is stored in ROM), we do not support runtime adaptation and statically generate program variants.
2. *Runtime adaptation:* For all other nodes we generate a DSPL using a subset of all features. We deploy only the features that are required for the used operating system, the hardware, special needs of a customer, and the roles a node can play.
3. *Binding units:* To avoid a high overhead at runtime and to reduce the number of possible variants for reconfiguration, we merge features into binding units when they are always used in combination.

7.3.2 Defining Binding Units

In Table 7.1, we show a sample assignment of roles for two types of node hardware and the corresponding binding units, which demonstrates the flexibility of our approach. The binding units are composed from the features of Figure 7.10. We depict sample configurations in Table 7.2. In our example, simple node hardware (*Simple* in Table 7.1) with highly constrained resources does not support runtime adaptation and can only be used for sensor nodes. We use a statically composed variant for these nodes (binding unit StaticSense in Table 7.1).

Hardware with less resource constraints (*Advanced* in Table 7.1) that supports reconfiguration at runtime is used for different roles. An advanced node is deployed with role *Positioning*, for computing the relative position of the node. A node unloads the feature when the position has been determined. If a *Sensor*, a *DataAggregator*, or an *AccessNode* is needed, an advanced node loads the required binding units. The node may also play different roles at the same time. For example, to

Binding Unit	Features
StaticSense	Positioning, Routing, Sensor, Radio, ST
Core	Routing, Radio, Wi-Fi, MT
Positioning	Positioning
Sense	Sensor
QueryProc	Access, SQL, Data, Storage
Aggregation	Aggregation, SQLite
Streaming	Streaming, BerkeleyDB

Table 7.2: Sample configuration of different binding units.

process a streaming query, a DataAggregator additionally loads the STREAMING binding unit.

We observe that our approach provides high flexibility with respect to possible deployment scenarios. We can define different feature configurations according to the used hardware at deployment time *and* according to required functionality at runtime. For example, a Sensor node uses different binding units but a similar set of features depending on the used hardware (Simple or Advanced). We can also define completely different binding units and feature selections according to used hardware, application scenarios, etc.

The feature selection highly influences the binary size of the generated node software. A variant that uses only static binding does not include any code for runtime adaptation. It has a binary size of 48 KB, which is only half of the size of a runtime-adaptable variant with the same features and a size of 104 KB. This overhead mostly comes from code of the infrastructure for runtime adaptation, which is independent of the number of features. The overhead is quite small compared to larger programs such as a node with stream processing, which has a binary size of 576 KB. Hence, our approach allows us to apply self-configuration also on resource constrained environments. Nevertheless, the resource requirements for runtime adaptation still limit its applicability and may require static binding when storage capacity is highly limited. With our approach, a user can choose at deployment time whether to use static binding or to support runtime adaptation.

7.3.3 Self-Adaptation

Adaptation Rules. For self-adaptation of a sensor node, we define adaptation rules within dedicated feature modules of the sensor network. For example, we place rules for activating and deactivating stream processing in feature STREAMING. The rules are thus included in a running program only if the corresponding feature is selected for dynamic binding. Based on the defined rules, a DSPL autonomously reconfigures itself according to the required features at runtime. In our scenario, a node loads the streaming binding unit when it receives a streaming query.

Reconfiguration of nodes is triggered by events spawned in monitoring code of the

DSPL. We implement the monitorings in distinct feature modules that extend classes of the application SPL to separate adaptation code from the SPL's implementation. For example, to activate stream processing, the monitoring code captures incoming queries and triggers the adaptation event when a streaming query is found. The corresponding rule adds a constraint for feature STREAMING (i.e., the feature must be included in a valid configuration). Another rule removes the constraint after all streaming queries have been processed. We do not directly remove the feature which would result in an unneeded reconfiguration when the feature is used again. A feature is only removed when it is excluded by other constraints or when other requirements such as limited working memory force to remove unneeded features. For example, we use a rule to unload the positioning feature when the position of the sensor has been calculated.

Reconfiguration. In Figure 7.11, we depict evaluation results for the adaptation process.[2] We analyzed the time needed for calculating whether an adaptation is needed and the time for reconfiguration. To show the benefits of statically optimizing the feature model, we compared reconfiguration of the same sensor node (1) using the original feature model of the SPL including all 55 features and (2) using the transformed feature model of the DSPL with 6 features (i.e., one feature per binding unit; cf. Sec. 7.1). In the diagram, we depict the time a node requires to process queries that are sent every 300 ms.

Begin of stream processing is triggered by streaming queries (denoted with b in Fig. 7.11), which results in a runtime adaptation to load binding unit STREAMING. The adaptation must be finished before the query processing can continue. The first streaming query is detected after 5 s. Loading the STREAMING feature takes 20–60 ms (note that we use a logarithmic scale) and increases the response time because the execution is continued after reconfiguration. Calculating the new configuration takes less than 1 ms. Assuming a minimal adaptation time of 20 ms, a node cannot reconfigure itself more than 50 times per second.

End of stream processing is denoted with (e). Instead of unloading the Streaming feature, a rule removes the constraint added before. Hence, all following adaptation events do not cause a reconfiguration. Nevertheless, the adaptation events increase the response time by about 0.3 ms when using the complete feature model with 55 features and 0.05 ms for the DSPL model with 6 features. This computation time is required for checking whether the node has to be reconfigured due to the context change. We use a SAT solver for this purpose. Compared to a reconfiguration that takes 20-50 ms, 0.3 ms is a very small overhead. However, it means that the node cannot handle more than about 3000 changes of the adaptation context per second even though no adaptation is needed. On an embedded device this would be much less due to limited computing power. By contrast, the node with the simplified feature model with 6 features requires only 0.05 ms for checking whether an adaptation is needed. This demonstrates the importance of reducing the variability

[2] For evaluation, we used an AMD 2.0 GHz CPU and Windows XP.

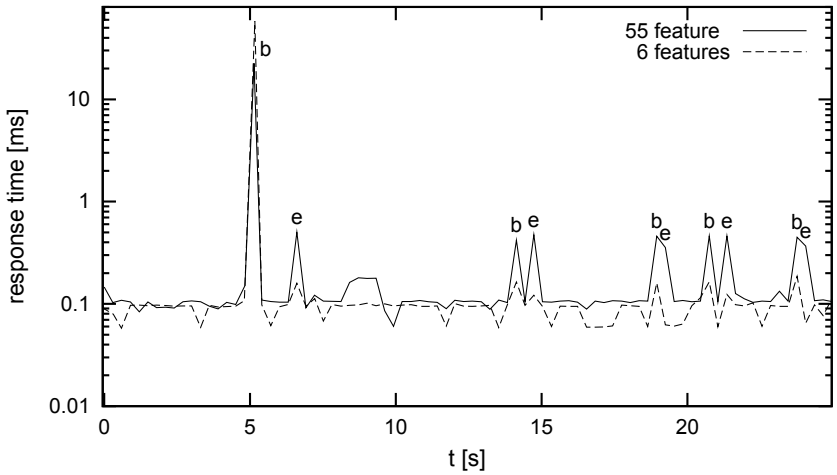


Figure 7.11: Response time (logarithmic scale) during reconfiguration of a query processing sensor network node using a feature model with 55 features and a simplified model with 6 features. Begin and end of stream processing are denoted with (b) and (e).

for runtime adaptation by optimizing the feature model.

7.3.4 Conclusion

In our case study, we combined static feature binding with support for feature-based runtime adaptation. We have shown that we can achieve autonomous reconfiguration by including the adaptation mechanism and the feature model into the running DSPL. By generating binding units, we further optimize runtime adaptations, as we discuss next.

Implementation-independent Adaptations. Using features to describe adaptations, we provide an adaptation mechanism that abstracts from the modules actually used for dynamic binding. Hence, we can generate binding units that fit to an application scenario and the execution environment while being able to reuse adaptation rules. When generating binding units, we transform the feature model and adaptation rules accordingly. We thus avoid a mapping of SPL features to binding units during adaptation at runtime.

Composition Safety. Using a feature model, we ensure that adaptations are correct with respect to domain constraints. As we have shown, this can be efficiently done at runtime before creating a variant by using a SAT solver. Furthermore, we can check if an adaptation rule is valid with respect to the feature model of the DSPL before runtime. With static type checking, we can even provide type safe runtime adaptations.

Resource Consumption. We provide an adaptation mechanism with low resource requirements (e.g., binary size, computing power) due to (1) customization of the adaptation infrastructure and (2) customization of binding units by removing unused code. The flexible size of binding units minimizes dynamic binding and enables static optimizations, as we have analyzed in Chapter 6.

Computational Complexity. We have shown that we can reduce computations for reconfiguration at runtime in two ways: (1) by avoiding unneeded adaptations and (2) by simplifying the computations for checking satisfiability by transforming the feature model according to actually available variability. The time required for computing a valid configuration is small compared to an actual reconfiguration even when using a feature model with 55 features. However, frequently checking whether a reconfiguration is needed can easily require more computing power than available. Hence, it is important to reduce the computation time by optimizing the feature model.

Including also non-functional requirements in these computations is a challenging task with respect to computational complexity [FHS+06]. Since the computation time to solve constraint satisfaction problems increases exponentially with the number of features [BRCT05, WDS09], it is even more important to reduce the number of binding units as far as possible. Our approach reduces the overall complexity and can be combined with CSP solvers to consider also numerical constraints (e.g., limited memory consumption) when computing an optimal configuration at runtime [SRK+08]. Further simplification is possible by caching the results of a SAT or CSP solver.

Constraint-based Adaptations. Directly modifying the configuration or an architectural model can result in unneeded reconfigurations and may cause configuration conflicts. Instead, we compute a new configuration based on the current configuration of a program and a set of requirements the new configuration has to fulfill. This avoids configuration conflicts and frequent reconfigurations, as we have shown in our evaluation.

7.4 Related Work

There are several approaches that use components and an architecture-based runtime adaptation as proposed by Oreizy et al. [OGT+99]. We further abstract from implementation details and use features for configuring a program at runtime. This allows us to reason about configuration changes at runtime at a conceptual level and to describe adaptation rules in a declarative way without taking the high-level architecture into account.

There are approaches that apply SPL concepts to develop adaptive systems and approaches that use feature-oriented concepts for modeling dynamic variability in

SPLs [FHS+06, HSSF06, LK06, TRCP07, CGFP09]. We aim at building a foundation for integrating them using features for SPL configuration *and* runtime adaptation. In the following, we compare our approach with respect to the most prominent approaches.

Some approaches describe dynamic variability in terms of features [LK06, TRCP07, CGFP09]. Lee et al. use a feature model to describe the variability (static and dynamic) of an SPL and they suggest to manually develop components (i.e., feature binding units) for implementing dynamic variability [LK06]. We decide *at build-time* which binding time to use and use features (i.e., not implementation units) for specifying dynamic configuration changes and for program validation. Furthermore, a component-based approach requires mapping features to components. We resolve the mapping before deployment by transforming the feature model and the adaptation rules accordingly.

Floch and Hallsteinsen et al. present with MADAM an approach for runtime adaptation that uses SPL techniques as well as architectural models [FHS+06, HSSF06]. They propose to model variability using component-based SPL techniques [HSSF06]. We use features for modeling variability and runtime adaptation to further abstract from the underlying implementation.

Cetina et al. use feature models to describe variability of an adaptive system [CGFP09]. To adapt a system, they modify a configuration by adding or removing features. This results in the problems discussed in Section 7.3.4, which we solve by using constraints to describe current requirements on a system. Furthermore, we seamlessly integrate SPL engineering and runtime adaptation by applying SPL concepts to adaptation code (e.g., adaptation rules) and by supporting static binding of features and merging of features into binding units. The result is a transformed feature model for runtime adaptation that represents only the required variability.

Morin et al. describe variability with a feature model and realize variability of the component model of an adaptive system with aspect-oriented modeling (AOM) [MBJ+09]. They use aspects to describe model adaptations and reconfigure a program based on changes of the model. By contrast, we operate on features that are not only implementation independent but also independent of the component model of a system. Hence, our approach can be combined with an approach for model adaptation. This allows us to validate a configuration before adapting the component model.

Safe composition is also important for component-based software development. For example, the Treaty framework combines verification of component assemblies using contracts and event-condition-action rules [CW10]. In contrast to these approaches, we aim at using feature models for validating configurations at runtime. Furthermore, FOP allows to check type-safety for an entire SPL before deployment. However, we do currently not provide any means for verifying compositions based on contracts, but we think that approaches for verification are orthogonal to a feature-oriented approach for composition.

We use feature modules for implementing adaptive systems, but our approach for feature-based adaptation may also be used with other implementation units such as

components or aspects [MBJ+09, CGFP09, LK06, TRCP07]. In this case, features are still used to describe adaptation changes. After a new configuration has been derived and validated, a corresponding set of components has to be determined. Some of the approaches above provide advanced capabilities for runtime adaptation not considered here (e.g., adaptation planning, state transfer, etc.). We argue that such advanced mechanisms are complementary to a feature-based solution and features can be used to improve these mechanisms.

7.5 Summary

Dynamic software product lines (DSPLs) combine concepts of adaptive systems and software product lines (SPLs) to provide dynamic variability. However, current DSPL approaches commonly use coarse-grained components to implement variability. This reduces customizability and thus limits applicability of a DSPL. We presented an approach that integrates traditional SPLs and DSPLs more closely. Based on a feature-oriented implementation of an SPL and a customizable adaptation framework, we *generate DSPLs* by statically composing features. As in traditional SPLs, we support fine-grained static customization for efficiency reasons; as in DSPLs, we provide adaptability at runtime by *generating* coarse-grained *dynamic binding units*. A dynamic binding unit is tailored to an application scenario by including only user-defined features. For runtime adaptation, we propose to describe adaptation rules in terms of features. We integrate this feature-based adaptation mechanism with our approach for generating DSPLs by transforming the feature model of an SPL according to the binding units of the generated DSPL.

By using features to define adaptation rules and configuration changes, our approach is independent of an SPL's implementation. Using a feature model, we efficiently validate program adaptations before modifying a configuration at runtime. Our integration of static binding and DSPLs reduces the overhead for dynamic binding. Furthermore, it avoids unneeded dynamic variability, which simplifies computations at runtime.

8 Concluding Remarks and Future Work

Product line engineering aims at providing a scalable approach for software composition by reusing assets across different programs. Feature modules are the units of reuse when implementing an SPL with FOP. However, current FOP approaches provide limited reuse because the binding time of a feature module depends on the implementation mechanism. To overcome this limitation, we presented extensions for FOP that allow programmers to implement an SPL with FOP and to choose the binding time *per feature* after development. We achieve this with code transformations for static and dynamic binding and with a mechanism that integrates both binding times. This allows us to abstract from the binding time and to apply FOP to several kinds of operating environments and application scenarios. We evaluated our approach with FeatureC++, a language extension for C++ that supports FOP. In the following, we provide a detailed summary and conclusions for the presented approach. Based on the introduced concepts, we suggest possible future work.

8.1 Summary

Chapter 2. We provided foundations on SPL engineering and feature binding in Chapter 2. After an overview about SPL engineering, we introduced SPL implementation techniques, described techniques to realize different binding times, and introduced the concept of staged configuration.

Chapter 3. In Chapter 3, we analyzed the benefits and drawbacks of different binding times. We reviewed approaches for SPL development with respect to their binding mechanisms. A combination of static and dynamic binding is often achieved by mixing different approaches for SPL development (e.g., the C preprocessor and components). This does not provide *binding time flexibility* for single features and thus limits reuse. This is solved by approaches that support static and dynamic binding for the same code fragments. However, most of these approaches do not aim at SPL development and they do not support safe composition with respect to a feature model. Furthermore, when binding features dynamically, each feature is usually bound as a separate unit, which increases the overhead for dynamic binding and the complexity of the dynamic composition process.

Chapter 4. We have presented extensions for the FeatureC++ code transformation process to avoid any overhead for feature composition. We evaluated FeatureC++ by refactoring Berkeley DB into an FOP implementation. We compared variability

implementation techniques of the C preprocessor and FeatureC++ with respect to customizability and resource consumption. We have shown that FeatureC++ can replace variability mechanisms used in Berkeley DB and allows us to generate programs with similar resource consumption. Nevertheless, both approaches exhibit a *functional overhead* if there are more features included in a program than actually required. This overhead is inherent to static binding and can only be reduced with dynamic binding. To achieve scalable static binding, we analyzed how clients can handle variability of generated components such as Berkeley DB. We introduced the notion of a *variable SPL interface* as the basis for subtyping in component families. Moreover, we presented mechanisms to handle component variability that we partly implemented in FeatureC++.

Chapter 5. In Chapter 5, we introduced a mechanism to transform the feature modules of FOP into modules that can be dynamically composed. This allows a programmer to implement the features of an SPL once and to decide which binding time to use after development. To achieve safe composition of features at runtime, we developed *FeatureAce*, a framework for automated dynamic feature composition. FeatureAce is included in a running program and supports dynamic binding according to an SPL's feature model. It thus avoids instantiation of invalid programs. In combination with static type checking [AKGL10], this allows us to achieve type safety for dynamic feature binding. Finally, we analyzed the compositional overhead caused by dynamic binding.

Chapter 6. To overcome the limitations of purely static and purely dynamic binding, we presented an extended approach that seamlessly integrates both binding times at the implementation *and* at the modeling level. Our integrated approach allows developers to statically generate *dynamic binding units* from a set of dynamically bound features. The binding units can be statically tailored and optimized and are bound as a whole at load-time or runtime of a program. By transforming an SPL's feature model according to the generated binding units, we achieve safety for composition of dynamic binding units. With two case studies, we evaluated the approach with respect to flexibility and resource consumption.

Chapter 7. In Chapter 7, we demonstrated applicability of our approach in the area of dynamic software product lines (DSPLs). Using binding units and an extended composition mechanism of FeatureAce, we are able to generate tailor-made DSPLs from an SPL. A generated DSPL supports rule-based runtime adaptation and self-configuration. We could show that dynamic binding units are appropriate to abstract also from complex binding mechanisms such as reconfiguration at runtime: programmers can describe adaptations in terms of features without the need to consider the actually used binding units. Compared to dynamic binding of individual features, our evaluation has shown that dynamic binding units reduce the

complexity of computing valid configurations and also of applying a new configuration at runtime.

8.2 Conclusion

We have shown that FOP is indeed appropriate to build highly customizable applications for use in different kinds of environments and application scenarios. In resource-constrained environments, the requirements on resource consumption and customizability are highly important and can be satisfied by FOP. Nevertheless, we have also shown that FOP may increase the effort for implementing variability. Compared to the C preprocessor, FOP may complicate variability implementation especially when fine-grained extensions below the granularity of methods (e.g., single statements) are required. In general, the effort for separating feature code into a feature module is higher than annotating the code fragment. Our observations are in line with other research in this area [Käs10]. We argue that this is the price that has to be paid for physical separation of features. In turn, the developer gets improved reuse because feature modules can be used in different contexts. However, this improved reuse is not always needed and an annotative approach may be better suited for SPL development than FOP. Consequently, FOP and annotations are complementary approaches for SPL development. Their combination is promising for further improvements of the SPL engineering process.

To overcome the flexibility problems of static binding, we provided an approach for generating binary feature modules that can be bound dynamically. Using FOP, we allow a programmer to decide after development whether static or dynamic binding should be used. To achieve safety also at runtime, we use an SPL's feature model to ensure instantiation of programs that are correct with respect to the feature model. While dynamic binding avoids the *functional overhead* of static binding, it introduces a *compositional overhead*. This often makes it intractable to apply purely dynamic binding when an SPL is decomposed into many small features.

Flexible Binding Times. Based on static and dynamic binding, we provided an approach that seamlessly integrates both binding times and combines their benefits. In summary, our approach provides means:

- to develop the features of an SPL using a single implementation mechanism that is independent of the binding time,
- to choose the binding time per feature after development,
- to generate dynamic binding units by composing multiple features to optimize resource consumption.

Our approach provides high flexibility when needed and still allows for fine-grained static customizations. Since it is a generalization of purely static and purely dynamic binding, it may replace such approaches. Compared to alternative solutions for

implementing static and dynamic feature binding, a dynamically bound feature is statically bound with respect to its binding unit. Our approach simplifies several aspects of SPL development by abstracting from the used binding time. For example, programmers can decompose an SPL with respect to units for reuse (i.e., the features of an SPL) and not with respect to the composition mechanism. At deployment time, they can decide which features to bind dynamically, which of these to bind at the same time via a single binding unit, and even whether to use dynamic binding at all.

With respect to resource consumption, we found a tradeoff between *functional overhead* caused by static binding and *compositional overhead* caused by dynamic binding. Finding the optimal binding time for the features of an SPL is a difficult task. Varying requirements on flexibility between different application scenarios further complicate the decision. Our proposal of generating dynamic binding units, allows an SPL developer to choose the binding time per feature even after development and for each application scenario individually. By combining a set of user-defined dynamically bound features in a binding unit we minimize the compositional overhead, which in turn reduces the adoption barrier for dynamic binding.

Towards Flexible Feature Composition. Flexible binding times are only a first step towards a flexible feature composition process. The flexibility can be increased by providing further binding times and other binding mechanisms. For example, FeatureC++ also supports dynamic binding by generating conditional expressions (i.e., `if`-statements; not discussed in this thesis), which reduces the compositional overhead but does not support independent deployment. Furthermore, we integrated static and dynamic binding at the level of the dynamic composition infrastructure: We developed FeatureAce itself as an SPL, which allows us to statically customize the binding mechanism as required per application scenario. By using FOP, FeatureAce supports static customization of adaptation rules, e.g., to choose between different runtime adaptation strategies. In summary, we propose several ways to improve the flexibility of feature binding:

- **Binding time flexibility:** pre-compile-time, compile-time, post-compile-time (e.g., link-time), load-time, runtime
- **Flexible binding mechanism:** code transformation, delegation, conditional statements
- **Flexible product instantiation:** manual vs. automated instantiation, configuration validation
- **Advanced mechanisms:** rebinding, self-configuration

In this thesis, we focused on selected binding times: static binding before compilation and dynamic binding at load-time or at runtime. These provide best optimization possibilities (static binding) and highest flexibility (dynamic binding). However, other binding times are interesting as well. For example, binding at link-time provides higher flexibility than static binding and may introduce a lower compositional

overhead than dynamic binding. Furthermore, other code transformations can be used (e.g., byte code transformation as supported by AspectJ), to reduce the compositional overhead of dynamic binding. Hence, there are many possibilities to further improve the flexibility of the feature composition process.

8.3 Future Work

There are several ways to integrate FOP with existing SPL technology. This is especially promising for improving scalability of software development, because features seem to be appropriate for describing large software systems using a high-level abstraction. As already indicated, future research may address several areas such as providing other binding times, improving consistency and safety of runtime adaptation, or optimizing non-functional properties of generated programs (e.g., in DSPLs). In the following, we focus on two challenges for further improving the integration of FOP and SPL engineering that are based on the presented ideas: (1) an integration of FOP with annotations and CBSE and (2) the application of FOP to the development of interdependent SPLs (a.k.a. *multi product lines (MPLs)* [RS10]).

8.3.1 From Annotative Approaches to Components

We observed that annotative approaches provide less reuse opportunities than FOP because annotations can only be reused in the context of the same surrounding code. However, also the feature modules of FOP provide limited reuse when compared to the concept of components with interfaces. Similarly, the effort for modularizing features with FOP is higher than annotating the feature code; but the effort is lower than developing components with an explicit interface. To summarize, the development effort and the reuse potential increases from annotations over feature modules to components. In the following, we describe how these three solutions can be further integrated to optimize the SPL development process.

Annotations and Feature Modules. Annotative approaches are well suited to implement variability for fine-grained extensions and when reuse across different SPLs is not needed. The reason is that the effort for separating feature code, especially single statements, into a feature module is higher than annotating a code fragment. As we could already show, some mechanism provided by the C preprocessor can be replaced by using FOP mechanisms. For example, we can use hook methods and method refinements to replace most macros. However, not all kinds of macros and uses of `#ifdef` statements can be replaced without significantly increasing the implementation effort.

A solution to solve these problems is to allow mixing annotations and code composition based on feature modules. That is, annotations (e.g., using *disciplined annotations* [Käs10]) may be used to *customize feature modules* if fine-grained extensions are needed and if modularization of features does not pay off:

- the annotated code is not intended to be reused in different contexts,
- separate development, deployment, etc. are not required, and
- when modularization effort, code complexity, maintenance, etc. are inappropriately high.

However, the decision whether to use annotations or feature modules has to be decided per code fragment. Hence, it must be further analyzed when to use an annotative or a compositional approach to avoid an inappropriate mixture of both approaches that degrades maintainability of an SPL.

Towards an Integration of CBSE and FOP. Components promise to improve reuse and to help solving the scalability problems of software development. However, CBSE also exhibits fundamental problems. As Biggerstaff pointed out, the scalability problem can be traced back to the *vertical/horizontal scaling dilemma* [Big98]: Increasing the functionality of a software component (i.e., vertical scaling) limits its applicability to a diminishing number of use-cases. The results are missing functionality, incompatibility, and unacceptable performance. *Horizontal scaling* can be used to overcome the limitations of vertical scaling by introducing component variants that satisfy requirements of different application scenarios. This in turn causes a combinatorial explosion of component variants and thus increases development and maintenance effort. Finally, it is hard to achieve both, vertical and horizontal scaling, with current programming paradigms.

We argue that combining components and FOP can reduce the scalability problems. To achieve this, we may either implement a component as a compound feature (i.e., composing it from several smaller features) or as an SPL itself (i.e., a *component SPL*). In contrast to developing monolithic components, this allows us to (1) customize a component by selecting the required functionality in terms of features and (2) to bind components statically or dynamically, e.g., to optimize resource consumption. This requires that:

- feature modules must support explicit (but optional) interface definitions,
- components are generated from a set of feature modules, and
- component generation supports arbitrary composition mechanisms.

In this thesis, we have shown that it is possible to generate dynamic binding units by composing feature modules, which is a first step to integrate CBSE and FOP. To fully integrate the approaches, we have to support generation of components with an explicit interface from a set of features. Hence, the relation between feature modules and components is similar to the relation between annotations and feature modules: we use annotations to customize feature modules and use feature modules to customize components. However, it has to be clarified when a component should be developed as an SPL and when it should be generated on demand, as we described for binding units.

In either case, we can provide a flexible mechanism for component composition using a generative approach. For example, we can switch between static and dynamic binding of components or can generate code to support different component models such as CORBA or EJB. Hence, components should not be developed for a predefined component model that is chosen before component development. Instead, we choose the required composition mechanism at deployment time.

Such an extension of FOP has to be accompanied by appropriate support at the conceptual level by representing components as compound features with an explicit interface and by transforming the feature model accordingly (cf. Sec. 6.2.3). This allows for checking consistency of compositions as we described for binding units. However, there are many more challenges on this way such as variability of component interfaces. To reduce the complexity due to variable component interfaces, a family of generated components must provide subtype relations and polymorphism between family members. This results in *component hierarchies* that improve reuse of components, as we described in [RSK10].

8.3.2 From Component SPLs to Multi Product Lines

Large software systems, such as distributed systems, often consist of several smaller subsystems or components that depend on each other. Developing the individual subsystems of a more complex system as SPLs results in a set of interdependent SPLs, which we call a *multi product line (MPL)* [RS10].

In this thesis, we provided a way to flexibly combine different composition mechanisms within a single SPL. For MPLs that are composed from multiple smaller SPLs, this approach must be extended to support different composition mechanisms across the SPLs. For example, some components may be statically composed to derive subsystems that are dynamically composed. FOSD and feature models allow us to describe and compose all SPLs in the same way and to handle dependencies among the SPLs. Features are especially promising for this task because they can be decomposed hierarchically and can be used to describe functionality that cuts across several SPLs. By generating the required composition mechanisms as we proposed for components above, we can furthermore implement an SPL independent of the finally used composition mechanism. This allows us to tailor the composition mechanism of a single SPL according to the requirements of the surrounding MPL and also with respect to a particular application scenario.

For further research this means several challenges at different levels. At the conceptual level, there have to be mechanisms to handle dependencies between SPLs. For example, feature modeling may be further extended with new concepts to support MPLs [RSKuR08, RS10, RSTS11]. At the implementation level, this means that developers have to consider constraints between SPLs to achieve safe composition and interoperability of an entire MPL [RSK10]. In the end, we face several composition stages: We compose components from multiple features, we use the components to build subsystems, and we build more complex systems from multiple subsystems. At every stage, we may use different composition mechanisms

and must support composition safety. Ideally, all the different mechanisms can be realized with a single flexible composition mechanism that is based on FOSD and feature models.

Bibliography

[AB06] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 59–68. ACM Press, 2006.

[AB11] S. Apel and D. Beyer. Feature Cohesion in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011. to appear.

[ABE+97] S. V. Adve, D. Burger, R. Eigenmann, A. Rawsthorne, M. D. Smith, C. H. Gebotys, M. T. Kandemir, D. J. Lilja, A. N. Choudhary, J. Z. Fang, and P.-C. Yew. Changing Interaction of Compiler and Architecture. *Computer*, 30(12):51–58, 1997.

[AG01] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. In *Proceedings of the Symposium on Software Reusability (SSR)*, pages 109–117. ACM Press, 2001.

[AGM+06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 201–210. ACM Press, 2006.

[AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, 2006.

[AH96] G. Aigner and U. Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer-Verlag, 1996.

[AK09] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[AKB08] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170. ACM Press, 2008.

[AKGL10] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-oriented Product Lines. *Automated Software Engineering – An International Journal*, 17(3):251–300, 2010.

[AKL08] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 101–112. ACM Press, 2008.

[ALMK10] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[ALRS05] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.

[ALS06] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 122–131. ACM Press, 2006.

[ALS08] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[AMGS05] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On Feature Traceability in Object Oriented Programs. In *Workshop on Traceability in Emerging Forms of Software Engineering*, pages 73–78. ACM Press, 2005.

[And00] R. Anderson. The End of DLL Hell. Technical report, Microsoft Corporation, 2000.

[ASB+09] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace. Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–17. University of Duisburg-Essen, 2009.

[Asp10] AspectJ Team. The AspectJ Development Environment Guide. Version 1.6.10., Available from http://eclipse.org/aspectj, 2010.

[Aßm03] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer-Verlag, 2005.

[BC90] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311. ACM Press, 1990.

[BCGS95] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the Symposium on Software Reusability (SSR)*, pages 27–37. ACM Press, 1995.

[BDC+89] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The Feature Interaction Problem in Telecommunications Systems. In *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62, 1989.

[BDG+94] D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, and J. Thomas. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, 1994.

[BHMO04] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 83–92. ACM Press, 2004.

[Big98] T. J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5(1):169–226, 1998.

[BJMvH02] D. Batory, C. Johnson, B. MacDonald, and D. v. Heeder. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 143–153. IEEE Computer Society, 1998.

[BM01] I. D. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 281—290. IEEE Computer Society, 2001.

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[BRCT05] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer-Verlag, 2005.

[BSBG08] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–32. IEEE Computer Society, 2008.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[CC04] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65. ACM Press, 2004.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CGFP09] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Using Feature Models for Developing Self-Configuring Smart Homes. In *Proceedings of International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 179–188. IEEE Computer Society, 2009.

[CHdM06] P. Costanza, R. Hirschfeld, and W. de Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC)*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer-Verlag, 2006.

[CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer-Verlag, 2004.

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multi-level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CHH09] A. Classen, A. Hubaux, and P. Heymans. A Formal Semantics for Multi-level Staged Configuration. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 51–60. University of Duisburg-Essen, 2009.

[CK03] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 50–59. ACM Press, 2003.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the European Software Engineering Conference*, pages 88–98. ACM Press, 2001.

[CKMRM03] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[CL01] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294. IEEE Computer Society, 2001.

[CRE08] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: Implementing Features with Flexible Binding Times. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 108–119. ACM, 2008.

[CW00] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10. Morgan Kaufmann, 2000.

[CW10] B. D. Claas Wilke, Jens Dietrich. Event-Driven Verification in Dynamic Component Models. In *Proceedings of the International Workshop on Component-Oriented Programming (WCOP)*, pages 79–86. Karlsruher Institut für Technologie (KIT), 2010.

[DG01a] K. R. Dittrich and A. Geppert. *Component Database Systems*. dpunkt.Verlag, 2001.

[DG01b] K. R. Dittrich and A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In *Component Database Systems*, pages 1–28. dpunkt.Verlag, 2001.

[Dij72a] E. W. Dijkstra. The Humble Programmer. *Communications of the ACM (CACM)*, 15(10):859–866, 1972.

[Dij72b] E. W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, pages 1–82. Academic Press Ltd., 1972.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dij82] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[DK02] A. Deursen and P. Klint. Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[DNS+06] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[Dre10] A. Dreiling. Feature Mining: Semiautomatische Transition von (Alt-) Systemen zu Software-Produktlinien. Diplomarbeit, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010. in German.

[DSGB03] S. Deelstra, M. Sinnema, J. V. Gurp, and J. Bosch. Model Driven Architecture as Approach to Manage Variability in Software Product Families. In *Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications*, pages 109–114. Springer-Verlag, 2003.

[EC00] U. Eisenecker and K. Czarnecki. Dynamische Komponenten und Architekturevolution. *OBJECTspektrum*, 2000(4):79–84, 2000. in German.

[Ern01] E. Ernst. Family Polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer-Verlag, 2001.

[Fav96] J.-M. Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 329–338. IEEE Computer Society, 1996.

[FHS+06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[FKAL09] J. Feigenspan, C. Kästner, S. Apel, and T. Leich. How to Compare Program Comprehension in FOSD Empirically – An Experience Report. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*, pages 55–62. ACM Press, 2009.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[FSSP07] W. Friess, J. Sincero, and W. Schroeder-Preikschat. Modelling Compositions of Modular Embedded Software Product Lines. In *Proceedings of the IASTED International Conference on Software Engineering (SE)*, pages 224–228. ACTA Press, 2007.

[GCH+04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[GFdA98] M. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 76–85. IEEE Computer Society, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gri00] M. L. Griss. Implementing Product-Line Features with Component Reuse. In *Proceedings of the International Conference on Software Reuse (ICSR)*, volume 1844 of *Lecture Notes in Computer Science*, pages 137–152. Springer-Verlag, 2000.

[GS05] W. Gilani and O. Spinczyk. Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays*, pages 94–109. Gesellschaft für Informatik, 2005.

[GS11] S. Günther and S. Sunkle. rbFeatures: Feature-Oriented Programming with Ruby. *Science of Computer Programming–Special Issue on Feature-Oriented Software Development*, 2011. To appear.

[GSC+03] A. F. Garcia, C. Sant'Anna, C. Chavez, V. T. da Silva, C. J. P. de Lucena, and A. von Staa. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In *Software Engineering for Large-Scale Multi-Agent Systems (SELMAS)*, pages 49–72. Springer-Verlag, 2003.

[GSD97] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of Database Management Systems based on Reuse. Technical Report ifi-97.01, Department of Computer Science. University of Zurich, 1997.

[HC01] G. T. Heineman and W. T. Council. *Component-based Software Engineering.* Addison-Wesley, 2001.

[HC02] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of Joint*

Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES), pages 38–45. ACM Press, 2002.

[HCN08] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, 2008.

[Her02] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of the International Net.ObjectDays Conference*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer-Verlag, 2002.

[HH04] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 26–35. ACM Press, 2004.

[HHPS08a] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[HHPS08b] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. International Workshop on Dynamic Software Product Lines (DSPL). In *Proceedings of the International Software Product Line Conference (SPLC)*, page 381. IEEE Computer Society, 2008.

[HK00] G. Hedin and J. L. Knudsen. On the role of language constructs for framework design. *ACM Computing Surveys (CSUR)*, pages 8–12, 2000.

[HKA10] F. Heidenreich, J. Kopcsek, and U. Aßmann. Safe Composition of Transformations. In *Proceedings of the International Conference on Theory and Practice of Model Transformations*, pages 108–122. Springer-Verlag, 2010.

[HKW08] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 943–944. ACM Press, 2008.

[HMPS07] C. Hundt, K. Mehner, C. Pfeiffer, and D. Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology (JOT) – Special Issue: TOOLS EUROPE 2007*, 6(9):417–436, 2007.

[HSSF06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 141–150. IEEE Computer Society, 2006.

[HZP+07] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, and S. Röttger. Enforceable Component-based Realtime Contracts. *Real-Time Systems*, 35(1):1–31, 2007.

[KA08] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 35–40. Department of Informatics and Mathematics, University of Passau, 2008.

[KAB07] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[KAL07] M. Kuhlemann, S. Apel, and T. Leich. Streamlining Feature-Oriented Designs. In *Proceedings of ETAPS International Symposium on Software Composition (SC)*, pages 177–184, 2007.

[Käs10] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010.

[KATS11] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011. to appear.

[KAuR+09] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190. Software Engineering Institute, 2009.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[KFFD86] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba. Hygienic Macro Expansion. In *ACM Conference on LISP and Functional Programming*, pages 151–161. ACM Press, 1986.

[KK07] M. Kuhlemann and C. Kästner. Reducing the Complexity of AspectJ Mechanisms for Recurring Extensions. In *Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 14–19, 2007. Published at the workshop Web site: http://www.softeng.ox.ac.uk/aople/.

[KKB07] C. Kästner, M. Kuhlemann, and D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Workshop on Refactoring Tools (WRT)*, pages 62–63. Technische Universität Berlin, 2007.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[Kru08] C. W. Krueger. The BigLever Software Gears Unified Software Product Line Engineering Framework. In *Proceedings of the International Software Product Line Conference (SPLC)*, page 353. IEEE Computer Society, 2008.

[KS08] H. Kegel and F. Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 431–440. ACM Press, 2008.

[KTS+09] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE Computer Society, 2009. Formal Demonstration paper.

[LAL+10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

[LAMS05] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-Oriented Software Development – FeatureIDE: An Eclipse-Based Approach. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 55–59. ACM Press, 2005.

[LAS05] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the*

East-European Conference on Advances in Databases and Information Systems (ADBIS), volume 3631 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, 2005.

[LB06] R. Lopez-Herrejon and D. Batory. From Crosscutting Concerns to Product Lines: A Function Composition Approach. Technical Report TR-06-24, University of Texas at Austin, 2006.

[LBL06a] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

[LBL06b] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77. ACM Press, 2006.

[LBN05] J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature-Oriented Designs. In *Proceedings of the International Conference on Feature Interactions (ICFI)*, pages 178–197. IOS Press, 2005.

[Lie86] H. Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223. ACM Press, 1986.

[Lip96] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[LK06] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. IEEE Computer Society, 2006.

[LLO03] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual Collaborations – Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, 2003.

[LM06] J. Lee and D. Muthig. Feature-oriented Variability Management in Product Line Engineering. *Communications of the ACM (CACM)*, 49(12):55–59, 2006.

[LST+06] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the International EuroSys Conference (EuroSys)*, pages 191–204. ACM Press, 2006.

[LW94] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[MBJ+09] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[MDK96] J. D. Mcgregor, J. Doble, and A. Keddy. Let Architectural Reuse Guide Component Reuse: A Pattern for ReUse. *Object Magazine*, 6(2):38–47, 1996.

[MFH01] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[MI06] I. Mahgoub and M. Ilyas. *Smart Dust: Sensor Network Applications, Architecture, and Design.* CRC Press, 2006.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

[MO04] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136. ACM Press, 2004.

[MSL00] M. Mezini, L. Seiter, and K. Lieberherr. Component Integration with Pluggable Composite Adapters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2000.

[MWC09] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. Software Engineering Institute, 2009.

[NAR08] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[NQM06] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested Intersection for Scalable Software Composition. In *Proceedings of the International*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 21–35. ACM Press, 2006.

[NR69] P. Naur and B. Randell, editors. *Software Engineering: Report of the Working Conference on Software Engineering, Garmisch, Germany, October 1968.* NATO Science Committee, 1969.

[NTN+04] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEEE Computer Society, 2004.

[OGT+99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-based Approach to Self-adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[Ost02] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2002.

[OT00] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 734–737. IEEE Computer Society, 2000.

[OZ05] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57. ACM Press, 2005.

[Par72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)*, 15(12):1053–1058, 1972.

[Par76] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1):1–9, 1976.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2):264–277, 1979.

[Par07] D. L. Parnas. Software Product-Lines: What to Do When Enumeration Wont Work. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–14. Irish Software Engineering Research Centre, 2007.

[PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, 2005.

[PKG+09] M. Pukall, C. Kästner, S. Goetz, W. Cazzola, and G. Saake. Flexible Runtime Program Adaptations in Java - A Comparison. Technical Report 14, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2009.

[PKS08] M. Pukall, C. Kästner, and G. Saake. Towards Unanticipated Runtime Adaptation of Java Applications. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 85–92. IEEE Computer Society, 2008.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[PSC09] M. Pukall, N. Siegmund, and W. Cazzola. Feature-oriented Runtime Adaptation. In *ESEC/FSE Workshop on Software Integration and Evolution at Runtime (SINTER)*, pages 33–36. ACM Press, 2009.

[pur04] pure-systems. Technical White Paper: Variant Management with pure::variants, 2004. http://www.pure-systems.com.

[RAB+92] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6):27–41, 1992.

[RALS09] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

[Ras03] A. Rashid. *Aspect-Oriented Database Systems.* Springer, 2003.

[RKS+09] M. Rosenmüller, C. Kästner, N. Siegmund, S. Sunkle, S. Apel, T. Leich, and G. Saake. SQL à la Carte – Toward Tailor-made Data Management. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 117–136. Gesellschaft für Informatik, 2009.

[RKSS07] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines:

A Case Study. In *Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 20–25, 2007. Published at the workshop Web site: http://www.softeng.ox.ac.uk/aople/.

[Ros05] M. Rosenmüller. Merkmalsorientierte Programmierung in C++. Diplomarbeit, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2005. in German.

[RS10] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. University of Duisburg-Essen, 2010.

[RSAS08] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM Press, 2008.

[RSAS11] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible Feature Binding in Software Product Lines. *Automated Software Engineering – An International Journal*, 18(2):163–197, 2011.

[RSK10] M. Rosenmüller, N. Siegmund, and M. Kuhlemann. Improving Reuse of Component Families by Generating Component Hierarchies. In *Workshop on Feature-oriented Software Development (FOSD)*, pages 57–64. ACM Press, 2010.

[RSKuR08] M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. ur Rahman. Modeling Dependent Software Product Lines. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 13–18. Department of Informatics and Mathematics, University of Passau, 2008.

[RSPA11] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel. Tailoring Dynamic Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM Press, 2011. to appear.

[RSTS11] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–20. ACM Press, 2011.

[Sam97] J. Sametinger. *Software engineering with reusable components*. Springer-Verlag, 1997.

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570, 1998.

[SB02] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[SC92] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 185–197. USENIX Association, 1992.

[SCK+96] M. Simos, D. Creps, C. Klingler, L. Levine, and D. Allemang. Organization Domain Modeling (ODM) Guidebook Version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas, VA, 1996.

[SE08a] K. Schmid and H. Eichelberger. From Static to Dynamic Software Product Lines. In *International Workshop on Dynamic Software Product Lines (DSPL)*, pages 33–38. IEEE Computer Society, 2008.

[SE08b] K. Schmid and H. Eichelberger. Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 63–71. University of Duisburg-Essen, 2008.

[Sim95] M. A. Simos. Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. *SIGSOFT Software Engineering Notes*, 20(SI):196–205, 1995.

[SKR+08] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, and G. Saake. Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 25–31. University of Duisburg-Essen, 2008.

[SKS+08] S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, and G. Saake. Generating Highly Customizable SQL Parsers. In *Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 29–33. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2008.

[SP10] S. Sunkle and M. Pukall. Using Reified Contextual Information for Safe Run-time Adaptation of Software Product Lines. In *Workshop on*

Reflection, AOP and Meta-Data for Software Evolution (RAM-SE), pages 1–6. ACM Press, 2010.

[SRA10] N. Siegmund, M. Rosenmüller, and S. Apel. Automating Energy Optimization with Features. In *Workshop on Feature-oriented Software Development (FOSD)*, pages 2–9. ACM Press, 2010.

[SRK+08] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring Non-functional Properties in Software Product Lines for Product Derivation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE Computer Society, 2008.

[Ste06] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 481–497. ACM Press, 2006.

[Str94] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1994.

[Str02] B. Stroustrup. C and C++: Siblings. *The C/C++ Users Journal*, 20(7):28–36, 2002.

[Sun07] S. Sunkle. Feature-oriented Decomposition of SQL:2003. Master's thesis, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2007.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley & Sons, 2006.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[TBD06] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 191–200. ACM Press, 2006.

[TBK09] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Society, 2009.

[TLSS10] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk. AspectC++: An Integrated Approach for Static and Dynamic Adaptation of System Software. *Knowledge-Based Systems*, 7(23):704–720, 2010.

[TOHSMS99] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.

[TRCP07] P. Trinidad, A. Ruiz-Cortés, and J. Peña. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *International Workshop on Dynamic Software Product Lines (DSPL)*, pages 51–56. Kindai Kagaku Sha Co. Ltd., 2007.

[TSH04] A. Tešanović, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS)*, pages 291–301. IEEE Computer Society, 2004.

[VBAM09] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. HotWave: Creating Adaptive Tools with Dynamic Aspect-oriented Programming in Java. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–98. ACM Press, 2009.

[vdS04] T. van der Storm. Variability and Component Composition. In *Proceedings of the International Conference on Software Reuse (ICSR)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2004.

[vGBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working Conference on Software Architecture (WICSA)*, pages 45–55. IEEE Computer Society, 2001.

[VN96a] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–369. ACM Press, 1996.

[VN96b] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *Object Technologies for Advanced Software, Second JSSST International Symposium (ISOTAS)*, volume 1049 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 1996.

[vO02] R. C. van Ommering. Building Product Populations with Software Components. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265. ACM Press, 2002.

[vO04] R. C. van Ommering. *Building Product Populations with Software Components*. PhD thesis, Rijksuniversiteit, 2004.

[WBA+10] T. Würthinger, W. Binder, D. Ansaloni, P. Moret, and H. Mössenböck. Applications of Enhanced Dynamic Code Evolution for Java in GUI Development and Dynamic Aspect-oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 123–126. ACM Press, 2010.

[WDS09] J. White, B. Dougherty, and D. C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

[Wit96] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

[XMEH04] B. Xin, S. McDirmid, E. Eide, and W. C. Hsieh. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, University of Utah, 2004.

[Zdu04] U. Zdun. Some Patterns of Component and Language Integration. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPlop)*, pages 1–26. UVK Verlagsgesellschaft mbH, 2004.

[ZJ03] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139. ACM Press, 2003.

[ZJ04] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205. ACM Press, 2004.

[ZSN07] U. Zdun, M. Strembeck, and G. Neumann. Object-based and Class-based Composition of Transitive Mixins. *Information and Software Technology*, 49(8):871–891, 2007.

[ZZM04] W. Zhang, H. Zhao, and H. Mei. A Propositional Logic-Based Method for Verification of Feature Models. In *International Conference on Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2004.