Automated Product Derivation for the CoCoME Software Product Line: From Feature Models to CoBoxes

Alexander Worret
University of Kaiserslautern

Submitted on March 26, 2009
Revised on May 6, 2009

Supervised by
Prof. Dr. Arnd Poetzsch-Heffter
and
Dr.-Ing. Ina Schaefer

Software Technology Group
Abstract

Constructing software products as members of a software product line (SPL), instead of usual single application engineering, offers many advantages like reduced time-to-market and centralized maintenance. Many domains of software engineering have, therefore, accepted the challenges of product line engineering in order to increase their efficiency and lower their maintenance costs. Yet, using a product line approach, as well as the respective process and technology, does not guarantee to actually gain from these possibilities. Constructing a product line requires a high up-front investment to later benefit from reduced development time and costs for each individual product. Creating a specific product from the specification of a product line, however, currently still requires a lot of manual work, especially during implementation, and can easily become so complex that balancing the up-front investment becomes a very difficult task.

Automated product derivation can help to address this issue, as a specific configuration of the product line can be created without additional effort. However, there is no clear and structured way for creating a product line allowing use of this automation.

In this work, we therefore create an exemplary software product line that supports automated product derivation from the Common Component Modeling Example (CoCoME). This product line is based on the CoBox component and concurrency model and demonstrates the steps required in the different phases in software development (modeling, design and implementation) for obtaining an SPL allowing automated product derivation. Later on, this experience is formalized to deliver first steps in creating a generic process for developing software product lines that are able to benefit from automated product derivation.
Acknowledgments

First of all, I want to thank my supervisors Prof. Dr. Arnd Poetzsch-Heffter, for giving me the opportunity to work on this interesting field of software engineering, and Dr.-Ing. Ina Schaefer, for the irreplaceable helpful feedback. Also, I want to thank Jan Schäfer, for the assistance with the CoBox compiler, as well as Christian Rensmann and Max Steiner for proof-reading this thesis.

Most importantly, I want to thank my parents for always giving me the freedom of choosing my own way, and providing the support allowing me to actually go it.
# Contents

1 Introduction and Motivation .................................................. 3
   1.1 Contributions of this work ............................................. 4
   1.2 Structure of this thesis ................................................. 4

2 State of the Art: Software Product Lines .................................. 5
   2.1 Definitions ........................................................................ 6
   2.2 Feature Modeling ............................................................. 8
      2.2.1 Feature Diagrams ..................................................... 8
      2.2.2 Feature Definition Language (FDL) ............................. 9
      2.2.3 Product Maps ......................................................... 10
   2.3 Variability management ..................................................... 11
      2.3.1 Variability management in requirements and design ........... 11
      2.3.2 Implementational variability management ....................... 13
   2.4 Characteristics of software product lines .............................. 20
      2.4.1 Comparison to other reuse approaches ......................... 20
      2.4.2 Chances of product line engineering ............................ 22
      2.4.3 Risks of product line engineering ................................ 22
   2.5 Summary ........................................................................... 23

3 Modeling the CoCoME Product Line .......................................... 24
   3.1 Introduction ....................................................................... 24
   3.2 Extending the example ..................................................... 24
   3.3 The Feature Model .......................................................... 27

4 Designing the CoCoME Product Line ........................................ 29
   4.1 The CoBox model ........................................................... 29
      4.1.1 CoBox characteristics .............................................. 29
      4.1.2 Benefits for the CoCoME product line ......................... 32
   4.2 CoBox-based design ....................................................... 33
   4.3 CoBox based design with variability .................................. 38
   4.4 Designing the product line .............................................. 41
   4.5 Summary ........................................................................... 47

5 Implementing the CoCoME Product Line .................................... 48
   5.1 CoBox-based variability management .................................. 48
   5.2 XVCL-based product line variability management ................. 49
   5.3 Implementing the product line ......................................... 52
   5.4 Summary ........................................................................... 57
## Contents

6 Formal Analysis of Software Product Lines 59  
6.1 Initial product line creation 59  
6.2 Product line life-cycle 69  
   6.2.1 Product line consistency 71  
   6.2.2 Additional functionality 72  
   6.2.3 Life-cycle tasks 77  
6.3 Summary 78  

7 Conclusion and Future Work 79  
7.1 Related work 79  
7.2 Future work 80  

A Exemplary source codes 82  
A.1 CoCoME product line constraint frame 82  
A.2 XVCL configuration file for configuring the CashDesk component 84  

Bibliography 86
1 Introduction and Motivation

Managing software reuse in an appropriate way allows benefiting from already performed tasks and long solved problems. Controlling reuse is, therefore, an important step in improving the maturity of the software engineering domain. Over the years, a lot of different approaches have evolved to, first, enable and, later on, improve the reuse of development knowledge and the resulting software artifacts. Since many companies specialize towards one or more specific domains, their product portfolio usually contains software products which overlap in terms of domain and functionality. Only a few products can be described as truly individual software. This leads to an increase in the popularity of product line engineering for managing reuse in the software development process. Product line engineering aims at constructing a series of products, which share a common domain, by analyzing and utilizing the commonalities and variabilities of these software systems [CN01]. This allows for structured reuse of created assets and centralized maintenance for a set of related products, thus potentially increasing efficiency and decreasing development costs significantly. Because of these promising benefits, many software developing organizations have embraced the product line approach. Due to this increasing popularity, a lot of effort was put into the creation of techniques, processes and technologies in the field of product line engineering.

However, a lot of this effort targets the early stages of software development, requirements engineering and design especially. Only some techniques target the implementation phase [A+02]. Although reuse is the driving force behind the development of software product lines, a lot of the implementation process is still very similar to the development of single applications. In several popular product line approaches ([ABM00], [B+99]), only the requirements and design documents are automatically tailored towards a specific product line member. Based on these documents the product is implemented manually, using shared and reusable software fragments. This allows creating an implementation which matches the corresponding requirements extremely well and allows for product-specific optimizations. However, this manual implementation also can easily negate many, if not all, of the benefits a software product line (SPL) might offer.

One possible solution to this problem is using automated product derivation. With this approach, products of the product line are not implemented on a per-product level. Instead, the product line is designed, implemented and maintained as a whole. This allows automatically deriving the products from a generic implementation. Previous work in the field of automated product derivation ([GS07], [McG05]) focused mainly on organizational and technical requirements. Only little effort was put into finding a clear and structured process for designing, implementing and maintaining a software product line supporting this automation. Without such a process, however, benefiting from the potential of the product line approach cannot be guaranteed.

For finding such a clearly structured way of developing SPLs with automated product derivation, in this work, we create an exemplary product line based on the Common Component Modeling Example (CoCoME) [H+08]. We analyze the steps required for modeling, designing and implementing the product line, such that automated product derivation can be used for obtaining any member of the SPL without product-specific development effort. The
experience from this development is transferred into a generic process and a set of declarative requirements, which have to be fulfilled in order to use the automated product line approach we present in this work.

1.1 Contributions of this work

In this work, we deliver first steps towards finding a well-structured way for constructing a software product line, which allows for automatically deriving its individual products. For this purpose, we extend the Common Component Modeling Example (CoCoME) to serve as a product line example and implement this example using the CoBox component and concurrency model [SPH08] developed by the Software Technology Group at Kaiserslautern University. Therefore, this work also delivers an objective feedback about the state of this component model. The central contributions of this work are

- an extension of the Common Component Modeling Example to an SPL. We create this exemplary software product line from a thorough analysis of the original example’s functional domain and capture the results of this analysis in a feature model.

- We create an extension to UML class diagrams allowing to design CoBox-based software product lines, and use it to create a design for the CoCoME product line. This design consists of a non-variable core system and a set of diagrams specifying deviation from this core in functionality and deployment.

- Based on the extended CoCoME design, we demonstrate how to implement a software product line designed with the introduced notation using frame technology and a two-step configuration process, such that automated product derivation can be used.

- We formalize the experience from modeling, designing and implementing the CoCoME product line, to create formal definitions for product line assets of different phases in product line engineering. With these definitions, we abstract the introduced product line approach on a theoretical level. Additionally, we use them for constructing a generic process, capable of both, creating and maintaining, a software product line with support for automated product derivation.

1.2 Structure of this thesis

In Chapter 2, we analyze the characteristics of product line engineering, how this specific approach can be compared to other reuse approaches and which methods and techniques are currently used for modeling, designing and implementing software product lines. In Chapter 3, we introduce and model an extended version of the Common Component Modeling Example which serves as an exemplary SPL. The following chapters target the different steps of software product line development. In Chapter 4, the CoBox component and concurrency model is introduced and analyzed, leading to a UML-based design notation for creating the design of the CoCoME product line. This design is implemented in Chapter 5 based on XVCL frames, such that automated product derivation can be used for the exemplary SPL. In Chapter 6, we formalize the development of the previous chapters to form a generic process for creating and maintaining an SPL supporting automated product derivation, using a set of declarative definitions and requirements. Finally, we distinguish our approach from related work and analyze possibilities for future work in Chapter 7.
2 State of the Art: Software Product Lines

Traditional software engineering creates a single software system from a set of requirements that are either received directly from a customer, without creative input from the developers, or created from the wishes described by a customer. Maintenance of a product requires altering the product according to modified requirements over a period of time. This approach results in a single development and a single maintenance process for each product. In case each product of a software development organization truly lies in a unique domain and does not share characteristics with other products, this is necessary and cannot be avoided. Usually, however, software development organizations specialize towards specific domains or even just one, single domain. In this case, starting a new development / maintenance process, when construction / evolution of a single application is required, can be highly inefficient. Much outcome of already performed requirements engineering, design and implementation tasks would be useful for several other products as well. Supporting a series of single applications in a specialized, shared domain, therefore, requires a lot of avoidable, redundant work.

Reusing the results of already performed tasks does not only allow shorter development time and with this lower development costs. It can furthermore also enhance the quality of these software products. A software product’s quality is based on the quality of its parts. Hence, in case a significant portion of the system is built by reusing software artifacts which already proved to be of good quality, the overall quality of the system is better, than if it was built without reuse [Mut02]. However, unstructured and ad-hoc reuse does not guarantee any of the described benefits. Many different reuse approaches exist and many of them don’t achieve the expected improvements (as pointed out by [A+02], [BGMW00], [Mut02] or [NTW05]). The most common reason for this failure lies in the lack of a well-defined method for organizing and managing reusable artifacts.

The software development approach of product line engineering, therefore, aims at shortening development time, as well as enhancing maintenance efficiency and quality for a series of similar software products through controlled reuse. Instead of constructing a series of single applications, in product line engineering, a shared architecture, a set of reusable artifacts and a reference model for creating products with the constructed architecture and artifacts are developed in a centralized family engineering process. This is done for all members of the SPL up-front, through a detailed analysis of the domain, the product line lies in (see Figure 2.1). The fragments constructed from this initial process target all phases of the software development process and are designed to ease the construction of a software product line (SPL) member. In a best case scenario, the construction of an SPL member later on simply reuses already constructed fragments, with only little additional design or implementation effort. The key in creating a product with few product-specific development effort lies in controlling the commonalities and variabilities of the product line and its members.

Figure 2.1 depicts the differences in the approaches of product line and single application engineering with optional reuse. Single application development constructs software simply out of the specific requirements of a customer. Reuse happens ad-hoc or is not used at all. Product line engineering, on the other hand, uses already created artifacts derived from the
State of the Art: Software Product Lines

Figure 2.1: Product line development (right) compared to single application development (left) according to [Mut02].

up-front family engineering process to match the requirements of a customer. Artifacts are created by domain analysis based on a series of potential, planned or already constructed products. Constructing an individual software product from these artifacts is called product derivation. This process is based on the reference model that is also constructed during family engineering. The model defines how the created assets can be used for deriving a member of the product line. To maintain the shared assets, feedback from the individual product derivation processes is used for iterating on both product line artifacts and reference model in the continuous process of family engineering.

To control the process of product line engineering and to improve software development using the methods of the product line approach, it is important to have a clear understanding of what product line engineering exactly is, how an SPL can be represented and how the design and the implementation of software products can benefit from these characteristics. In Section 2.1 we, therefore, provide definitions for the most important terms of this work. In Section 2.2 we discuss feature diagrams, the Feature Definition Language and product maps as different possibilities for creating a feature model and why explicit feature modeling is important for product line engineering. Afterwards, the usage of these models and their information in generic product line artifacts is shown for the different phases of software development in Section 2.3. Finally, in Section 2.4 we analyze the chances and risks of product line engineering, after comparing it to other reuse approaches.

2.1 Definitions

Since many terms in the context of software product lines are used differently in several publications, in the following, we provide the exact definitions for the terms used in the context of this work.
Software Product Line  In [CN01] a software product line is defined as a series of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. In result, these software systems share common characteristics (commonalities) and are distinguished from each other by their remaining characteristics (variabilities). Product line engineering aims at controlling these commonalities and variabilities by reusing a common, generic architecture and product core to produce the individual products with reduced development time and costs. To ensure that the information about commonalities and variabilities is captured appropriately, product line engineering uses feature models.

Feature model  A feature model describes a software product line and its products as sets of features. In [vDK02], a feature is defined as "a characteristic of a concept that is relevant in some context". In the context of this work, the concept in question is a set of software products. Hence, a feature is defined as a characteristic of at least one software product line member that is relevant for the development of this system or the product line itself. Features may represent functionality that is included in a product, which is their most common use. However, a feature may also refer to the platform of the software system, non-functional requirements or other aspects that may characterize a member of the SPL. The feature model declares the commonalities of all products and the possible variabilities for any member of the product line, as well as additional constraints between features. This includes for example choices between several features, mutual exclusion of features or other restrictions.

Feature modeling is the process of creating a feature model in the family engineering process constructing a product line. This usually means that a set of features is identified, analyzed and their constraints structured according to expert domain knowledge and experience. The specific set of features represented by a given member of the product line is called the configuration of the product line for this product. Since the feature model describes all features and their underlying relationship, the model declares all possible configurations of the SPL.

Product Derivation  Product derivation describes the process of creating a member of the software product line from shared product line assets. This process may introduce new features into the feature model before the actual derivation.

A product line usually aims at a high amount of reuse and, therefore, shares a common architecture and other reusable assets. This means that the derivation of a product has to deal with correctly identifying already existing and matching assets and setting them up according to the configuration of the new product and the product line's reference model. As these assets usually do not provide all functionality of a particular product, on the implementation level it is often required to implement additional, product-specific code. This code is not introduced into the shared code base, architecture or reference model. If new features were introduced into the feature model, fragments for implementing it must be created, brought into the shared code base and the reference model needs to be modified.

Automated product derivation, in contrast to manual product derivation described above, does not allow project-specific code modifications. Instead, all additional code is brought into the reference model and code base, so that every configuration of the product line can be derived using an automated process specified by the product line's reference model. The result of this automated configuration process is an executable software system.
2.2 Feature Modeling

One of the biggest advantages of product line engineering over other reuse approaches is the up-front knowledge of possible reuse scenarios. During the initial family engineering process all supported configurations of the product line are identified. Hence, the assets of the product line can be manufactured to explicitly support all possible configurations.

Of course, for this advantage to benefit development, information about the commonalities and variabilities of possible system configurations has to be captured and documented in an appropriate way, so that developers may access it in all steps of product line engineering. This is done by creating a feature model expressing the supported configurations during family engineering. For a product line with automated product derivation this is even more important. Automated product derivation does not allow product-specific code alteration, hence, the possible configurations of the feature model directly relate to the number of possible software systems that can be derived. Over the last years several possibilities for feature modeling have evolved. In the following, the most commonly used methods and techniques are described and analyzed towards their advantages and disadvantages.

2.2.1 Feature Diagrams

Feature Diagrams, originally proposed in [KCH+90], are graphical representations of product line member characteristics and their corresponding constraints. A feature diagram represents the entity to be modeled as a collection of hierarchically structured features. In this work’s context the entity to be modeled is a member of a software product line. The features belonging to the entity can either be mandatory or optional. Furthermore, it is possible to model choices in which either one or several features can be chosen. Additionally, each feature may be composed of several sub-features, which may use all of these constraints. This must not introduce any cycles to the graph and results in a tree of features, with the entity to be modeled as its top node.

A feature diagram for modeling a simple car is shown in Figure 2.2. The central entity ‘Car’ is modeled as a set of the mandatory features ‘carBody’, ‘Transmission’, ‘Engine’ and ‘Horsepower’. Those features are marked mandatory because of the filled circle connecting them to the top node. This indicates that every car requires these features, meaning they
build the commonalities of all cars. Additionally, a car may have the feature 'pullsTrailer'. This feature is optional, since its circle is not filled. The 'HorsePower' feature is a simple classification, as it can be represented by exactly one of 'highPower', 'mediumPower' and 'lowPower'. In a valid configuration exactly one of those classifications must be chosen. This 'one-of' choice is represented by the unfilled triangle at their parent feature 'HorsePower'. In analogy, the transmission of the car can be either 'automatic' or 'manual'. Feature 'Engine' is different, however. It represents a choice between 'electric' and 'gasoline'. Choosing both features (for a hybrid drive car) is also allowed and, therefore, represents a 'many of'-choice, instead of the other 'one of'-choices. Such a choice is depicted by a filled triangle at the corresponding parent feature.

Since the original proposal in 1990 several others have extended the concepts with the aim of adding additional layers of expressiveness ([vGBS01], [RBSP02], [Rie03]). The refinements and modifications presented in [RBSP02], for example, allow to directly state 'n of'-choices by adding multiplicities to the model, similar to the ones used in UML class diagrams. This, however, does not extend the expressiveness of the model, as proven by [BHST04]. It merely makes the models improve in readability. Despite several extension attempts, the most important disadvantage is still unsolved today: Feature diagrams are not well suited for big feature models, as these tend to take up too much space.

2.2.2 Feature Definition Language (FDL)

The Feature Definition Language [vDK02] is a text-based feature modeling approach. The language is used to compose atomic expressions with additional constraints into higher-order features. This results in a tree with the top node being the entity to be modeled. This makes the FDL very similar to feature diagrams. Instead of representing the features and the constraints between them in a graphical manner, however, FDL uses expressions derived from a context-free grammar.

The benefits of this textual and formal representation are:

- Especially for electronic data transfer, text representations are easier to handle, compared to pictures and diagrams.
- The formal nature of the representation allows algebraic methods to work with the expressions.

These algebraic methods allow for obtaining optimized representations of a product line through normalization and reduction of trivial descriptions. This can help in the initial domain analysis, when the domain of an SPL is built from the requirements of a set of products and the "natural" constraints of the domain are hard to understand or unknown.
Such a set of planned configurations leads to a disjunctive normal form. This easy to obtain representation, can then be used for automatically creating the domain’s constraints through algebraic reduction.

On the downside, a feature model described in FDL, can usually not be understood as easily as the graphical representation of a feature diagram. This can seen in Listing 2.3. The example is semantically identical to the one used in Section 2.2.1 for demonstrating feature diagrams. Yet, understanding the semantics of the example requires more time.

2.2.3 Product Maps

Product maps [Mut02] represent a product line by stating its existing or possible members, as well as the features belonging to each product. A feature map of all possible products is semantically equivalent to a feature model given by an FDL statement in disjunctive normal form. However, usually feature maps do not cover all possible products, but only the ones that are in development or at least planned. Product maps are different from the previously mentioned feature modeling possibilities, as they only list products and their features and not explicitly model the constraints that lead to these products.

A possible product map for the exemplary car product line modeled in the previous sections is shown in Figure 2.4. The lines corresponding to the 'carBody', 'Transmission', 'Engine' and 'HorsePower' features show that these features are present in all products and are, therefore, considered to be commonalities. With only few products, however, it is possible that features are present in every product by chance and not by design. Due to this fact, and due to the missing explicit modeling of the features’ constraints, product maps by themselves are not a sufficient feature model. In combination with another feature model, however, product maps can give a good overview of planned or already constructed products and may also indicate a trend towards features that are used more often than others. This information is usually not captured by other feature models.
2.3 Variability management

Feature models capture the commonalities and variabilities of a software product line. Managing this variability is very important for the success of product line engineering and needs to affect all development phases. During the initial family engineering process, a set of reusable artifacts is constructed. These artifacts are used for constructing all members of the product line. During product derivation, the variability of the generic assets has to be resolved. The generic artifacts have to be tailored to the individual product requirements. These requirements are expressed by the feature configuration corresponding to the product to be created. Of course, different software development phases require different generic assets, which are created during family engineering.

To study these different artifacts a common, short example is used. The example library software system is proposed in [BMG01] and manages loaned books for a library. Apart from the general functionality of managing book loans, the software optionally supports reservation of books. In the following section we analyze how this very basic optional functionality is handled in different development phases. We study how this variability is built into generic product line artifacts and how the variability can be resolved, to configure the generic artifacts into specialized ones.

2.3.1 Variability management in requirements and design

When deriving a product from the product line, the variability captured by a feature model needs to be resolved in order to receive a functional software product. Popular product line methods, like KobrA [ABM00] or PuLSE [B+99], use generic documents and decision models to manage variability on the requirements and design level. Decision models capture the modifications that are required to transform generic requirements and design documents, into specific ones, matching the product that is currently developed. An example for a generic requirements document can be seen in Figure 2.5. The document defines use cases for the exemplary library system to support book loans and reservations. As book reservations are not supported by all configurations the corresponding parts of the document are marked with a "variant" comment.

Figure 2.6 shows an example of a decision model for resolving the variability of the library software system’s requirements document in Figure 2.5. If the variability with ID 2 for example is resolved to not allow book reservations, the reservation use case UC2 is removed from the document. Otherwise, the use case remains in the document and its "variant" comment is removed.

Decision models can also be used for managing variability in generic software designs, like UML class diagrams for example. In this case, the decision model can be used to alter multiplicities in the diagram or to introduce new classes. Examples of this approach can be found in [ABM00] or [SHJ03]. The KobrA [ABM00] approach, for example, uses UML diagrams with additional "<<variant>>" stereotype annotations to indicate variability in the design. An example of this can be seen in Figure 2.7. During product derivation a decision map is used to resolve the variability of the annotated design. An exemplary decision map for resolving the variability in Figure 2.7 is shown in Figure 2.8. Should the configuration to be derived not include reservation support, the Reservation class, the ReservationManager class, as well as the association between Account and Item, is removed. If reservations are supported, only the stereotype annotations are removed and the association becomes
Use Case UC1: Loaning a book

Priority: high
Trigger: Customer wants to loan a book
Pre-conditions: none
Post-conditions: Book is loaned to the customer, loan is logged in the system
Actors: Customer, library worker
Steps:
1)...
2) !variant! System checks if the desired book is marked as reserved
3)...

Use Case UC2: Reserving a book

Priority: high
Trigger: Customer wants to reserve a book
Pre-conditions: none
Post-conditions: Book reservation is logged in the system
Actors: Customer, library worker
Steps:
1)...

Figure 2.5: Simple variable requirements document.

<table>
<thead>
<tr>
<th>Artifact</th>
<th>ID</th>
<th>Name/Question</th>
<th>Variant Point</th>
<th>Resolution</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
<td>Are users allowed to</td>
<td>Use Case UC1</td>
<td>yes</td>
<td>remove !variant!</td>
</tr>
<tr>
<td>Requirements</td>
<td>2</td>
<td>reserve books?</td>
<td>Use Case UC2</td>
<td>no</td>
<td>remove step 2</td>
</tr>
<tr>
<td>Requirements</td>
<td>2</td>
<td>reserve books?</td>
<td>Use Case UC2</td>
<td>yes</td>
<td>remove !variant!</td>
</tr>
<tr>
<td>Requirements</td>
<td>2</td>
<td>reserve books?</td>
<td>Use Case UC2</td>
<td>no</td>
<td>remove use case UC2</td>
</tr>
</tbody>
</table>

Figure 2.6: Decision model resolving variability of a library software system in the requirements phase.

Figure 2.7: Simple variable UML class diagram.
an invariable part of the system’s design. The \texttt{<<variant>>} stereotype allows modeling simple Boolean "all or nothing" variability in UML diagrams. Refinements of this notation, e.g. [SHJ03], define additional stereotypes for allowing more fine-grained constructs. The combination of decision models and variable documents is a common approach for handling the design and the requirements of a product line.

2.3.2 Implementational variability management

While decision models work well for requirements documents and formalized design descriptions, on the implementational level they are not well suited. On the requirements or design level, resolving variability usually can be handled through manipulation of use-cases, class members or multiplicities. Describing the manipulation of a generic source code artifact, however, can be both confusing ("replace lines X to Y of while-loop between lines Z and W with ...") and error-prone, as malfunctions might exist only due to an unusual modification of the generic code.

Therefore, managing and resolving variability on the implementation level has to be taken care of differently. Generally speaking, the presence or absence of a specific feature in the current configuration may require code alteration on a generic base artifact in different ways:

- Add new code to the base artifact.
- Remove code from the base artifact.
- Change the code contained in a base artifact.

In order to achieve the above modification possibilities several approaches exist. The most commonly used approaches are introduced in this section.

Dynamic

Dynamic variability management includes all techniques resolving variability at runtime [RSSA08]. Examples for this are data-driven if- or loop-conditions. It is possible to resolve all
variability of a software product line dynamically, by providing an appropriate configuration file to the system. In this case, the code for all product line members is identical, and the configuration of each product is solely achieved by modification of this configuration file. This approach, however, imposes a lot of limitations and does not suit all product lines. Especially requirements targeting non-functional characteristics, like performance for example, are very hard to resolve dynamically. Resolving variability dynamically requires checking the configuration of the system at various occasions. This runtime performance overhead potentially hinders fulfilling performance constrains.

In most cases, it is, therefore, not reasonable to resolve the variability of a software product line dynamically. It can, however, still be a good choice for some aspects of variability management, especially if some variability should be left for the customer to resolve.

**Static**

Static variability management refers to all techniques used to resolve variability before runtime [RSSA08]. This can be during compilation or even before compilation. Since static resolving does not cause any performance overhead at runtime, it is more important to product lines than dynamic resolving. Hence, numerous techniques are in use today [AG01]. The choice of which methodology to use depends on the programming language chosen for implementing the SPL and the tool support available. Additionally, some particular features can be resolved more efficiently with certain methods compared to others. In case the features of a product line are used only for adding functionality to a small core system, the techniques for handling implementational variability have to support code additions only and not necessarily code modification and removal.

The following paragraphs introduce the techniques available today and highlight their benefits.

**Conditional compilation** The term conditional compilation applies to all techniques which modify the source code of a software product (or here: a member of a software product line) based on rather simple Boolean conditions. Such technology is available for many platforms and languages. Probably the best known example for such a tool is the C/C++ pre-processor. The pre-processor sorts out marked code fragments from compilation in an up-front pass. After pre-processing, it is impossible to distinguish the processed resulting code from manually constructed code.

In Listing 2.9 an advantage of the pre-processor over other approaches is demonstrated. Since pre-processing definitions get handled by simple text replacement, constants like the number of iterations in a loop for example can be handled very elegantly. This would undoubtedly be possible with a variable holding the number of iterations as well, however, having a replaceable constant in the source code allows for optimizations during compilation. Optional loop unrolling for example is very easy in the above loop, once the constant is replaced. With a variable, a possible modification of the variable at runtime has to be analyzed before unrolling the loop.

The example in Listing 2.10 shows a possible implementation of a C++ class. This example is based on the sample library software introduced in Section 2.3.1. Based on the atomic definition of RESERVATION the optional method reserveBook and the class variables required are included. The example shows that even for such a simple example, the notion of what exactly is modified by the variability of a product line is not obvious.
2 State of the Art: Software Product Lines

Listing 2.9: Exemplary generic class showing an advantage of conditional compilation

```c++
#define NUMBER_OF_CHILDREN 3

class Parent {
    Child[] children = new Child[NUMBER_OF_CHILDREN];
    ...
    for (int i=0; i<NUMBER_OF_CHILDREN; i++)
        children[i] = ...;
}
```

Listing 2.10: Exemplary generic class implementation with use of pre-compiler in C++

```c++
#define RESERVATION YES // or NO */

class LoanManager {
    #if RESERVATION == YES
        ReservationManager* reservationManager;
    #endif
    public:
        int loanBook(Book* book, Account* account);
        int returnBook (Book* book);
        #if RESERVATION == YES
            int reserveBook(Book* book, Account* account);
        #endif
    }
    ...
    #if RESERVATION == YES
        int LoanManager::reserveBook(Book* book, Account* account) {
            return reservationManager->reserveBook(book, account);
        }
    #endif
    ...
```
With a Boolean expression, conditional compilation is able to either add functionality or remove functionality. However, the pre-compiler does not allow more sophisticated constructs for modifying present functionality besides the demonstrated atomic expressions and simple operations for combining a series of atomic expressions. Therefore, the source code for the implementation of a generic component, in a product line with much variability, can easily become very confusing and hard to understand. This becomes even more drastic in case several conditions for handling variability overlap. Although these disadvantages demonstrate that conditional compilation was not originally designed to be used for large-scale reuse, as in a product line, "with additional support documents it is still considered a valid choice to be used" [Mut02].

Object-orientation One of the fundamental concepts in object-oriented programming, inheritance, aims at code reuse by refining and adapting a subclass based on the implementation of the corresponding superclass. With inheritance, all attributes, methods, inner classes and methods are copied from the super- to the subclass, which additionally allows the addition of new attributes and methods, as well as the adaptation of inherited methods. This guarantees that a subclass provides at least the functionality that would be required for a subtype.

Listing 2.11 contains the generic source code for a simple class. This class is to be adapted according to the library software example using inheritance. The subclass presented in Listing 2.12 inherits the full functionality of its superclass. Additionally, it expands functionality by adding the reserveBook method, along with the required attribute, and adapts the behavior of the loanBook method. It should be noted that the code of the sub-class is semantically identical to the conditional compilation example with RESERVATION defined as YES, at least for programming languages in which inheritance does not automatically lead to sub-typing. The unspecialized super-class, however, corresponds to the previous example with RESERVATION defined as NO.

In conditional compilation all code for one class is contained in one source code document. In contrast, in object-orientation the modifications applied by a sub-class are usually separated from its super-class containing the base code. This can be, both, an advantage or a disadvantage. If the modifications applied are rather complex and require many additional code lines, holding them in a separate document can improve the readability of the base document. On the other side, very short and specialized modifications are usually hard to understand without their base code.
Frames  The frame technology [Bas96] is a macro language originating from the COBOL
programming language. It has been adapted and made available for many other platforms
and languages. Frames are used to inject and/or modify code at predefined positions based
on the source code of an already constructed system. Although the frame language defines
only five basic commands and was not designed for product lines, its capabilities for product
line reuse are a lot more sophisticated than the previously mentioned methods, as code can
be added, removed and modified very elegantly.

The basic commands of frames are:

- **BREAK**: Prepares conceptual code blocks for future modification.
- **COPY/INSERT**: Reuses an already defined frame but inserts code at defined code
  blocks.
- **REPLACE**: Replaces code blocks or variables.
- **SELECT**: Directly implements alternative code blocks.
- **WHILE**: Performs modifications based on a regular pattern.

The usage of these commands should be introduced in more detail with a short example.
In Listing 2.13 a generic class implementation for the exemplary library software using frame
technology is shown. The class is generally defined as code blocks for private attributes and
public methods. The implementation of the `loanBook` method furthermore consists of three
code blocks. The first one guards the execution of the method’s core, which is located in the
second block, and checks required pre-conditions. A final and third block is executed once the
method core was successfully run. These blocks can later be replaced, modified or extended.
The code resulting from the unmodified example is identical to the code resulting from of the
conditional compilation example with `RESERVATION` defined as `NO`.

To adapt the above generic class implementation for a specific product line configuration,
the frame given in Listing 2.14 injects code into it. The **COPY LoanManager** statement at the
top of the frame indicates that the LoanManager implementation is to be used as the base for

```cpp
class LoanManagerWithReservation : LoanManager {
    private:
        ReservationManager* reservationManager;
    public:
        int reserveBook(Book* book, Account* account);
    }
    int LoanManagerWithReservation::loanBook(Book* book, Account* account) {
        if (reservationManager->isReserved(book)) return 1;
        super.loanBook(book, account);
    }
    int LoanManagerWithReservation::reserveBook(Book* book, Account* account) {
        return reservationManager->reserveBook(book, account);
    }
    ...
```

**Listing 2.12**: Exemplary use of inheritance for adapting the generic object-oriented code.
\%Frame LoanManager

class LoanManager {
    private:
    BREAK privateAttributes {};

    public:
    BREAK publicMethods {
        int loanBook(Book* book, Account* account);
        int returnBook(Book* book);
    };

    int LoanManager::loanBook(Book* book, Account* account) {
        BREAK preLoanBook {
            if (book->isLoaned()) return 1;
        };
        BREAK mainLoanBook {
            ...
        };
        BREAK postLoanBook {};
    }

    ...

Listing 2.13: Exemplary generic class using frames.

\%Frame LoanManager—with—Reservation

COPY LoanManager {
    COPY LoanManager {
        INSERT—AFTER privateAttributes {
            ReservationManager* reservationManager;
        };
        INSERT—AFTER publicMethods {
            int reserveBook(Book* book, Account* account);
        };
        INSERT—BEFORE preLoanBook {
            if (reservationManager->isReserved(book)) return 1;
        };
        ...
    };
    int LoanManager::reserveBook(Book* book, Account* account) {
        return reservationManager->reserveBook(book, account);
    }

    ...

Listing 2.14: Exemplary frame for modification of previously defined code.
code extension. Based on the original implementation the modified implementation can be obtained by inserting a new private attribute at the privateAttributes break, a new public method at the publicMethods break and an additional condition that has to be met before the loanBook method may be executed. This code is injected at the at the preLoanBook break point. The resulting code is identical to the conditional compilation class with RESERVATION defined as YES.

This macro language allows very complex and powerful variability management. Often it is even possible to express the desired code modification in different ways. An optional code block can, for example, be expressed either as a code block which is moved when not needed, as a code block that is inserted if needed or as a SELECT statement holding both the code block and an empty BREAK block. This flexibility allows to create all code modifications that are possible using inheritance. The reverse, however, is not possible, as inheritance is unable to remove code. Like inheritance frames use an explicit separation of modification and base code.

**Aspect-orientation** Aspect-orientation is derived from the finding that concerns, which cannot be specifically mapped to one particular module of a system, are hard to address when developing software with traditional object oriented methods. These cross-cutting concerns are expressed and encapsulated in aspects. As cross-cutting concerns may easily manifest from features (as these may target any characteristic of the system), aspect-oriented programming (AOP) is of natural interest for implementing a product line. Additionally, since aspect-orientation modifies object-oriented code according to the implementation of aspects, it can also be used to modify functional code and manage product line variability.

An aspect consists of a set of pointcut definitions and a set of code modifications. The pointcut definitions specify possible places in the generic source code at which modifications can be applied. A modification contained in the aspect can select one of these pointcuts to specify a requirement that has to be matched to result in modified code. The actual code modification is performed in a pre-compilation step called aspect weaving. During this process the modifications of included aspects are applied at all places in the code that match their corresponding pointcut definition.

Working with AOP should again be demonstrated by using the short library software example. Listing 2.15 contains the generic Java implementation of the mentioned example. To
modify the code and add support for book reservations AspectJ can be used. The aspect implementing the additional code, as required by the library software, is shown in Listing 2.16. It defines a pointcut definition for the LoanManager.loanBook method. The aspect furthermore uses this pointcut to insert code before the execution of the method. Additionally, the aspect furthermore defines a new method reserveBook to the LoanManager class. After aspect weaving, this method could be used as if it were an original member of the LoanManager class. This makes aspect-orientation able to perform all necessary tasks for modifying the library software example.

The pre-compilation step and the explicit separation of inserted code and base code make aspect orientation very similar to frames. However, unlike frames, aspects do not allow removing code from a generic class. Also, frames specify their insertion points in the base code, whereas aspects specify pointcuts together with the actual modifications. This allows creating additional modifications without modifying base code.

2.4 Characteristics of software product lines

Software product line engineering is just one among many different reuse approaches in software engineering. However, some characteristics of product lines make this approach stand out compared to others. After analyzing the process of product line engineering and the techniques for constructing generic product line artifacts, in the following we discuss these characteristics, their chances and associated risks.

2.4.1 Comparison to other reuse approaches

Domain engineering The domain engineering reuse approach (as analyzed in [Mut02]) aims at providing different software products by capturing an entire functional domain. This is done through domain analysis and the construction of a domain-specific reference model during an initial domain engineering phase. The goal of this reuse approach is that any potential product within the captured domain should be constructible from the reusable software artifacts designed during domain analysis.
This, however, can be rather problematic. Often the bounds of a domain, and with this also the bounds of the reference model, are very hard to grasp. Creating a reference model from a too small view of the domain may exclude important characteristics from it and may fail to address important issues. This would lead to massive changes to the model while constructing future products, thus, revoking the value of the previous domain analysis task.

On the other hand, the domain can also easily become too big. Often small characteristics of a domain are safe to ignore for a large portion of possible products. Building a too big reference model including these niche characteristics, may be possible only with enormous effort. Since these unusual characteristics are required for only few (potential) products, an overly big reference model may hurt cost effectiveness just as much, or even more, as a reference model that is too small. This problematic task of finding the right bounds for an application domain makes the concept of domain engineering not very practical concerning reuse.

Product line engineering shares many characteristics with domain engineering. However, instead of capturing an entire application domain, product line engineering builds its domain of possible software products from the requirements of a set of already planned (or already existing) products. This leads to the domain spreading just as wide as any planned products require, leaving out any characteristics that would be irrelevant to the product line anyway. But still, scoping the domain of the product line has to be done carefully. If a new product is required to be included in the product line, but is not contained in its corresponding domain, the domain has to be extended. Having to expand the domain may prove just as cost-intensive as a mis-planned domain engineering approach.

Component-orientation  In component-oriented development software is constructed using software components. These components are reusable software fragments built for a specific component model. This model defines specifications each component has to fulfill. Usually, components can be seen as at least one runtime object, able to provide a specific service to its environment through a well-defined interface masking its implementation. Components are largely reused from other development projects or bought from third-party developers. The main tasks of a developer using component-orientation to construct a software product, is to configure these services according to the current project’s needs and to construct glue code, which allows using these services in the context of the software product to be constructed. The underlying paradigm of component-orientation can be described as the vision that developers should be able to construct any software product using components, as long as the repository of reusable, well-known and configurable components is big enough to satisfy the requirements imposed on the new software.

It should be noted that the definition of a component is heavily overloaded, as mentioned by [A+02], in computer science generally and in software engineering especially. The definitions used in literature often differ based on the component model used, as well as on personal opinion and experience of the authors. The common ground for many definitions, however, is that a component is a reusable, loosely coupled and, therefore, replaceable software artifact. It encapsulates implementation and is usually exposed to the environment through a set of interfaces.

Although component-orientation was generally accepted and several, eventually domain-specific, component models were created and are being used, the reuse approach never got quite the success it aimed for and its aspired market of prefabricated software, although
existing, is not as large as originally envisioned. This is due to problems which component orientation was unable to resolve as yet.

Often components are constructed during the life-cycle of a specific system with the desire for reuse, but without a specific reuse-scenario in mind. This may result in components that are not designed for being configured into different environments of other products. It can also result in components with so many parameters that explicitly grasping the context, in which they should be used, may no longer be possible.

Non-functional requirements of a component can also easily cause new issues, when brought into another development or runtime context. The additional need for stability and security in the system a component was originally built for, may be irrelevant for a new project. Additional security on its own usually is not a problem. However, as a fixed characteristic of a reusable component it may hurt other non-functional requirements, like performance for example, in the context of the new product. It is, therefore, very important to understand what effects reusing a specific component has on the new product. Only the components that fully satisfy all of the current project’s requirements should be used. Additionally, combining components from different component models may be impossible, if the respective systems they originated from targeted different and incompatible platforms.

In the context of product line engineering components can be used for a) creating a logical structure and b) providing functionality that is required in a slightly different context at various occurrences in the overall system. The usual component-oriented problems described above can all be avoided in this specialized context. Since the domain bounds of an SPL are fixed, the reuse scenario in which a component needs to work (and has to be configured for) is already known to full extend. Hence, each implementational fragment for a software product line can be constructed to fully match all possible reuse scenarios. Also, as parts of one single product line, the product line components also target the same platform and are, therefore, technically compatible. However, a widening of the domain can still be very problematic, as it might require changing a lot of components to reflect this change of domain bounds.

2.4.2 Chances of product line engineering

With well-structured processes for both family engineering and product derivation, product line engineering can reduce product development time significantly, as designing and implementing the product can be achieved with high reuse. In case new requirements do not manifest in form of new features, automated product derivation even allows obtaining the new product without any additional design or implementation effort. Also, maintenance effort of the product line can directly be propagated to maintaining the individual products. A fixed security defect in the common core for example automatically benefits all members of the product line.

2.4.3 Risks of product line engineering

Probably the biggest risk in product line engineering comes from a potential widening of the domain bounds, similar to the problem described for domain engineering. Extending the number of possible products by adding new features to the feature model may impose a huge set of change requirements necessary to reflect the new feature model in the components and their architecture. In this case, the risk of loosing many of the benefits described above exists and it may be a valid option to construct the product to be not a member of the software product line instead.
Another risk comes from one of the most powerful chances: centralized maintenance. Combined with the purely technical requirement of having to propagate changes from the central component repository onto every project, implementing changes on central components has the risk of breaking existing (and working) products. Apart from the product line as a whole, individual products need to be maintained, as well. This means that the context in which a shared component is used can change over time. If this modified context is not considered when adapting a central asset it may break an existing product. If the product in question has reached the end of its life-cycle and is not supported anymore this is not a problem. In general, however, every product still has to work with modified core components. This results in additional effort to make sure that maintenance of the product line assets does not affect individual products where this is not desired.

2.5 Summary

In this chapter, we have introduced the general process of product line engineering. Additionally, we considered the need for explicit feature modeling, especially for automated product derivation, and how the variability of these models can be managed in the generic product line assets for different development phases. The following chapters now discuss the feature model, design and implementation of an exemplary software product line.
3 Modeling the CoCoME Product Line

Explicitly capturing the variability of a software product line is crucial for all phases of software development and for the success of this reuse approach. The following section introduces the exemplary software product line used for studying automated product derivation. First, Section 3.1 introduces the origin of this product line, the Common Component Modeling Example. After extending the exemplary requirements in order to create a product line in Section 3.2, in Section 3.3 a feature model based on these requirements is constructed.

3.1 Introduction

The Common Component Modeling Example (CoCoME) is an exemplary software system, created for a software modeling contest held by the "Gesellschaft für Informatik" (German Computer Science Association) in 2006/2007 to compare different component-based modeling approaches. The software system is a good example for component-based software development.

The exemplary system described in [H+08] represents a cash desk system used for payment transactions in a supermarket or similar stores. During each transaction the cashier enters the products a customer wants to buy into the system using either a scanner or entering their respective product IDs using a keyboard. Entering the price of a product manually is not allowed, as this would exclude automated inventory management. After entering all products the customer pays the final product value by either cash or credit card. A store consists of an arbitrary number of these cash desks. Each of those is connected to the store server, holding product data like prices and inventory stock for each store. The store client, which is also connected to the server allows manipulation and analysis of product data and also to switch a set of cash desks into an express mode. While a cash desk is in express mode, which is indicated by a light display at each desk, the amount of products a customer may buy is limited and credit card payment is disabled. Additionally, all stores are connected over an enterprise server, which allows for a set of statistics to be created.

Figure 3.1 depicts the basic structures of the cash desk system with the connected peripheral equipment like bar code scanners, credit card readers, etc., as well as the store and enterprise server structure. The images are taken from the original example proposal in [H+08].

3.2 Extending the example

In order to use CoCoME as a product line example the original proposal has to be seen as one possible configuration of an SPL. The full product line, however, contains a set of possible cash desk systems serving varying purposes and shop scenarios. To create the functional domain of such an SPL, in the following section we extend the original CoCoME system and analyze the general requirements of a cash desk system.
Figure 3.1: Structure of the Common Component Modeling Example system [H⁺ 08].
A generic cash desk system allows a cashier to perform payment transactions. The cashier enters products into the system using a number of input controllers. Handling product input to the system can be done using a bar code scanner or a keyboard. In case a scanner is not present, a keyboard has to be connected to the system for entering the respective product IDs manually. On the other hand, a scanner is required should no keyboard be present. If both input options are available they may be used interchangeably. Based on the product IDs entered by the cashier, the cash desk polls a central store server to obtain the product’s data set, calculates the overall value of customer’s desired products and displays the information to the cashier. Entering a price manually is not supported, as it would result in inventory stock data becoming uncontrollable. The system allows paying the goods by different payment options. These payment options are cash, credit card, a prepaid card or an electronic cash system like Maestro or the German ”EC” system. The keyboard used for product input is also required for cash payment, as the cashier needs to enter the value of the cash money she received from the customer. This allows the system to calculate and display the difference. Allowing cash payment additionally requires a cash box being connected to the cash desk.

In case the system is used for selling goods, which don’t have their price calculated on a per piece base, as in a supermarket selling fruit and vegetable for example, there are two options: One possibility is to let each cash desk be provided with an additional scale to measure the weight of a product. In this case the product data in the store inventory indicates, if a specific product needs to be weighed. Alternatively, weighing the goods may happen at a specialized weighing machine in the respective part of the shop. This machine allows specifying the product type, weighing the goods, producing a bar code label, which is to be attached to the goods, and sending the bar code information with the corresponding price to the inventory on the store server.

The original CoCoME system uses an enterprise server to create a connection between several stores and to allow performing a number of enterprise-oriented tasks, like product transfer and synchronization between stores. In a general scenario, however, this is not always required. If a store is not part of a bigger franchise, the enterprise server can be omitted. Additionally, small businesses don’t require a high number of cash desks. For these stores having just one cash desk is often sufficient. In that case, this one cash desk may also take over the role of the store server and the enterprise server, if it is present.

In many cases the cash desk system replaces an existing software system. To allow reuse of already available server hardware, the operating system platform for enterprise and store server should be variable. This, however, does not affect the cash desk clients. These clients are very specialized, so that pre-existing client hardware is most likely incompatible. Additionally, the costs for client hardware are relatively low compared to server hardware. Therefore, in contrast to server application(s), keeping client-side software simple and invariable regarding the underlying platform is of more use to the overall development than potential client hardware reuse.

Another optional part of the system is self service for cash desks. At self service desks customers process their payment on their own, without the assistance of a cashier, which speeds up their payment process and saves costs for the store. To control this process, the products entered by a customer are weighed and compared to the weight of the product that is archived on the inventory. In case weighed goods are supported by the system, self service desks may be equipped with a camera system. This camera analyzes which type of product is potentially on the scales and displays the possible choices to the customer. As a final requirement, the express mode described in the original CoCoME system should not be a mandatory part of the product line, as it would not be of use in all scenarios.
3.3 The Feature Model

In this section we develop a feature model of the software product line based on the extended example described above using feature diagrams. The model is introduced in a two-step process. The first step creates a basic model in which all the identified features are grouped and structured hierarchically. In the second step, the semantical constraints between the product line features are addressed and the feature model is altered accordingly.

For modeling the CoCoME product line we selected feature diagrams from the different possibilities described in Section 2.2. The reasons for this decision are that product maps are not a sufficient feature model as they lack capturing feature selection constraints. This leaves feature diagrams and FDL. Those approaches are semantically identical, so the choice has to be decided based on their individual advantages and disadvantages. Variability of the CoCoME product line is ‘slim’ enough, so that the graphical notation of a feature diagram is still well arranged. Additionally, the model does not have to be transmitted electronically and the feature constrains of the product line are easy to understand, so there is no need for the algebraic bonuses of FDL. These factors clearly favor the use of feature diagrams.

Based on the requirements of the extended example the model has to include variability for different payment options, several input options, several optional features like the use of an enterprise server or express mode support and a choice for the server platforms. Structuring these features yields the feature diagram shown in Figure 3.2. It is important to note that the CashDeskSystem, the root of the model, is not just the application for the individual cash desks, but the whole software system including server applications. The system mandatorily requires a non-empty set of payment options and at least one of two input options, keyboard and scanner input. The mandatory server platform of the system has to be (exactly) one of Linux, Windows and Macintosh. All other features, like express support and self service support, are optional features that are not present in all configurations of the product line. Support for weighed goods can include weighing at the cash desk by a cashier or separate weighing by the customer. At least one of those two options has to be chosen, should weighed goods support be included in the system. These features represent all functional and platform characteristics of the product line members.

While the basic requirements are met with this model, a series of additional constraints are not yet captured. To begin with, a self service desk requires a scanner, as customers should
not have to deal with manually entering product IDs. Additionally, self service support is unable to work with cash payment. Since the feature model covers the entire system this does not mean that cash payment has to be excluded in case self service support is required, it simply means that at least one of the other payment options is required as well. Express mode support, on the other hand, clearly requires cash payment, as the (temporary) exclusion of other payment options is the major step in speeding up the payment process. Furthermore, a single desk system may neither be combined with self service support, as this would expose store server functionality to customers, nor may it be used with support for the express mode, as speeding up payment with only one cash desk is entirely up to the cashier. To model these additional constraints, the original feature diagrams of [KCH+90] use declarative text annotations. However, as described in [RBSP02], it is also possible to include these constraints in the diagram itself.

In Figure 3.3 we use the in-diagram notation for additional constraints, which results in our final feature model of the cash desk system product line. To model the constraints that are not represented in Figure 3.2 the optional single-desk feature needs to be split into two mutually exclusive features for single and multi-desk, of which one must be chosen in every valid configuration. The multi-desk feature is represented only implicitly in the previous model. In the new model, however, it has to be represented explicitly to serve as parent feature for weighed goods, self service and express mode support features. Additionally, non-cash payment options have to be grouped by a new corresponding feature. This expresses that self-service support requires one of these payment options at least.
4 Designing the CoCoME Product Line

Managing the variability of a software product line is important for all steps of the development process. This includes the design phase, the subject of this chapter. Additionally, the concepts applied in the implementation phase should already be known and understood at design time. Otherwise, designs could potentially be introduced that are impossible to implement, due to the available technology not being powerful enough. The CoCoME product line implementation in Chapter 5 uses the CoBox component and concurrency model. The first section of this chapter, therefore, introduces the general concepts of CoBoxes. Afterwards, an extension to UML class diagrams for designing CoBox-based systems is introduced. This notation is extended to also capture product line based variability management. After these design methods are established, they are used to design the CoCoME software product line.

4.1 The CoBox model

The CoBox model [SPH08] is an object-oriented, data-centric component and concurrency model developed at the Software Technology Group at Kaiserslautern University. A CoBox is a runtime component consisting of a (non-empty) set of runtime objects. These runtime objects are furthermore used to execute a set of tasks, of which at most one can be active at any given time. A task is active for as long as it is not finished executing code, unless it willingly suspends to wait for a specific event or simply to wait for a period of time. Hence, inside of a CoBox all code is executed sequentially. This results in CoBoxes being not affected by usual concurrency problems, e.g. race-conditions. A CoBox communicates with other CoBoxes outside of its own box with asynchronous messages. This design results in several characteristic properties that highlight CoBoxes in the field of distributed and concurrent programming.

4.1.1 CoBox characteristics

Controlling invariants Regular concurrency programming in object-oriented languages usually requires manual thread synchronization based on primitives such as locks or semaphores to ensure that a concurrent method invocation does not eventually break an object’s invariants. In a CoBox, however, at most one task is active at any time. The time at which a task may be suspended is a design choice of the developer. Therefore, CoBoxes do not require any additional synchronization methods for ensuring invariants, as long as they hold at the specifically designed points in time at which a suspended task may become active. The following example demonstrates this:

The CoBox class in Listing 4.1 defines a simple invariant, namely that the boolean variable invariantMet has to equal true. After the execution of the private method n() the invariant does not hold anymore. As method m() calls n(), at least temporarily the current object is brought into an undesired state. In unsynchronized concurrent programming any method call that happened after the execution of method n() and before the invariant could be restored
Designing the CoCoME Product Line

```java
public class C {
    private boolean somethingHappened = false;
    private boolean invariantMet = true;
    // invariant: invariantMet == true

    public void notify() {
        somethingHappened = true;
    }

    private void n() {
        ...
        invariantMet = false;
    }

    public void m() {
        // invariant holds, do something
        n();
        // invariant does not hold anymore
        ...
        invariantMet = true;
        // invariant corrected, wait for notification to continue
        while (!somethingHappened) {
            JCoBox.yield(5000);
        }
        // got notified, continue execution...
    }
}
```

**Listing 4.1: Simple CoBox example.**

by method m(), would result in a method invocation with unfulfilled pre-conditions. This could potentially leave objects of this class in virtually any state. The CoBox semantics, however, guarantees that this cannot happen. By calling method m() (from outside of the CoBox) an inactive task is spawned in the respective CoBox. After a series of other task have been executed in this CoBox, the task designated to execute m() becomes active. During the execution of the task no other tasks may become active, unless it is designed by the developer. In the example this is happening with the invocation of the yield command. It (repeatedly) suspends the currently active task for 5 seconds and allows other tasks to be executed in the current CoBox until, ultimately, one of this tasks executed the notify() method.

**Advanced scheduling controls** A CoBox communicates with other CoBoxes via asynchronous message invocations. These asynchronous method calls have two results: First, they spawn a new task in the receiving CoBox to execute the called method and, second, yield a future object to the calling task. This future allows extended control over the result of the method call and over the point in time at which it is required. This is demonstrated by a short example in Listing 4.2.

In the example CoBox C uses another CoBox D to receive an integer value from method m. While the asynchronous method call (denoted by the exclamation mark instead of a period) does not result in the integer value itself, it yields a future object to obtain an integer value at some later point in time. During the execution of m() in the D CoBox, C may calculate other things until the result of the method call is required. At that time C may call the get() method on the future object. This either yields the result of the method invocation or blocks C’s thread until the result is available from D. During this time no other tasks are executed in CoBox C.
4 Designing the CoCoME Product Line

```java
public cobox class D {
    public int m() {
        ...
    }
}

public cobox class C {
    private D d;
    ...

    public void performTask() {
        Fut<int> f = d.m();
        // do something else
        ...
        // wait for the result of async. method call
        int i = f.get();
        // do something with 'i'
    }
}
```

Listing 4.2: Simple CoBox example.

Synchronous calls to methods are therefore (semantically) equivalent to making an asynchronous call and immediately executing `get()` on the resulting future. So, in the above example a synchronous call `d.m()` would be semantically identical to using `d!m().get()`.

**Message ordering** In the original CoBox proposal in [SPH08] the selection which of the many suspended tasks to activate was left non-deterministic. Meanwhile, the current implementation allows to optionally specify an ordering of messages. This ensures that a sequence of messages, given the calling and called CoBox are identical, remains in-order during activation, thereby also decreasing the need for manual synchronization.

**Value and immutable objects** Apart from scheduling and managing tasks, CoBoxes also structure memory at runtime by acting as containers for runtime objects. To ensure invariants of CoBox classes and, generally, guard the state of CoBoxes, all inner objects of a CoBox are protected by the surrounding box. This mean that even if a reference to an inner object of CoBox A is handed over to CoBox B, scheduling method calls that are made on this object is still subject to control of its owning CoBox A.

Of course this protection should only apply to objects where this is necessary. The state of immutable objects for example is not modified after their creation, so scheduling any of their (read-only) methods would clearly be a waste of (computation) time. For this purpose, classes may be annotated to be immutable. Taking care of the class actually being immutable is still a developer’s task, yet, this allows turning off unnecessary scheduling and consistency checks.

Another annotation for a class is the value annotation. Objects of a value class are automatically deep-copied each time a reference to such an object leaves the scope of their belonging box. This can happen when a reference to such an object is used as a parameter of a method call or when it is the result of a method. In contrast to the regular object-oriented call-by-reference behavior, this allows creating classes that are called by value.

This automated behavior created through the annotation of usage type annotations is explained in another short example. In Listing 4.3 different usage type annotations of classes in the CoBox model are shown. Objects of class F, which is declared to be a value type, hold
three different attributes: another value object, an immutable object and a regular object. An object of type F is held in the owning object of a CoBox and can be the result of method `getF()`. Every time a client executes this method, f is deep-copied, as it is labeled as a value type, and the copy is returned instead of f. In the example, deep-copying f results in a new object f2 with the following attributes:

1) `f2.c` holds a deep copy of `f.c`, since class C is also a value class,
2) `f2.d` holds a reference to `f.d`, since class D is immutable and needs no additional protection,
3) `f2.e` holds a reference to `f.e`, since class E is a regular class.

Access to attribute e is guarded by the CoBox of the server, as the reference originated from this box.

### 4.1.2 Benefits for the CoCoME product line

The cash desk product line is a highly concurrent and distributed system. Regular object-oriented mechanisms for manual thread synchronization tend to be overly complicated, as well as hard to understand and read. Therefore, they are not the first choice for implementing such a system. CoBoxes on the other hand are easy to understand and fully support concurrency and distribution. Especially the latter is very well supported by the model. Whether a set of CoBoxes is running in one process, in different processes on the same physical machine or on a wide network of machines virtually has no influence on the design of the individual components or the whole software system. Since the deployment of the components is variable in the design of the cash desk product line, this CoBox characteristic is very useful for developing the system.

The value type annotation furthermore supports the construction of a distributed system. An object of such a type is automatically copied, each time it needs to be referenced from outside its designated CoBox. More specifically, a copy of a value object is created in the
Designing the CoCoME Product Line

CoBox which requires a reference to it. In case of the object being a message parameter, the copy is created locally for the called CoBox. If it is the result of a method invocation, the objects' copy is created locally for the calling CoBox. This can help to decrease the number of remote invocations substantially, as further method invocations on such an object can always happen without remote communication.

Another helpful characteristic is the ordering of messages. Consider the following scenario: A cash desk is connected to a keyboard, which is used for entering product IDs into the system. The keyboard can be enabled and disabled, based on the state of the current payment process. The state is modeled through a boolean value in the keyboard class. Due to the non-deterministic nature of thread scheduling, messages may appear on the keyboard in a different order than they originated from the cash desk. A possible sequence of messages on the cash desk is ["Disable", "Enable", "Enable", "Disable"]. This sequence, without further data, would leave the keyboard in the "disabled" state and might lock the whole system in a state that is waiting for input from a deactivated device. In this simple example, the problem can be solved rather easily by attaching time stamps to the messages. Based on these time stamps the keyboard could track the time of the last processed message and ignore messages, which were sent before. However, in a more realistic scenario, the state of a component is much more complex than a boolean value. Unlike the keyboard in this example, the future state of a component usually depends on its previous state. In this case, a time-stamp-based decision between processing or dismissing a message altogether is not an option. Instead, the correct state of the component has to be built from an ill-ordered message sequence of unknown length. This is undoubtedly a very complicated task. The CoBox semantics, on the other hand, guarantees that a sequence of messages with identical origin and destination boxes are executed in the order they were sent. This makes it much easier to keep the whole system in a consistent state, as it does not require any additional timing and synchronization mechanisms.

4.2 CoBox-based design

After introducing CoBoxes as the basis for the design of the CoCoME product line, the next step towards implementation is to create a CoBox-based design for the SPL. CoBoxes are designed to target concurrency and distribution on a programming language level. Therefore, until now, only little effort was put into creating a design method for the model that can be compared to regular object-oriented design approaches, e.g. the UML. The following section therefore introduces a CoBox-based design notation and uses it to design one configuration of the CoCoME product line.

CoBox diagrams Until now, the only graphical representations of CoBox design were introduced in [SPH08]. An example is depicted in Figure 4.4. These diagrams represent a snapshot of the system during runtime and present a set of objects together with their owning CoBoxes and the references that are currently existing at that point in time. Of course these diagrams are not sufficient to describe the design of a large software system. However, pre-existing design approaches like UML do not capture the bounds of CoBoxes. Therefore, these two diagram types should be combined in order to describe the a CoBox-based system.

In JCoBox, the current Java-based implementation of the CoBox specification, programs without any CoBoxes are regular Java programs. Therefore, the design notation that is introduced within this chapter follows the same principle. It is based on UML class diagrams.
and a diagram that is not using any extensions will in fact be a regular UML class diagram. Where the UML has to be extended is the notion of CoBoxes structuring the system.

Structuring a software system using CoBoxes is expressed through a rounded rectangle with the name of the CoBox at the top. Each CoBox requires a class inside of this CoBox with the same name. This is the owning class of this CoBox and is created by implementing a CoBox class. A CoBox class may hold additional regular and CoBox classes to indicate that instances of the surrounding CoBox class hold instances of these nested classes. This can be seen in Figure 4.5.

In contrast to regular UML class diagrams the additional semantics for runtime structuring may require some classes being present in more than one CoBox. Having a fully designed class in a diagram several times clearly is a lot of redundant information. For this purpose the design language allows hiding information, if it is not important for a specific context. Adding an asterisk (a '*' symbol) to the class name indicates that the information contained within is not the full specification of the class. Such a specification is called partial. Partial class specifications may only display attributes and methods that are declared to be publicly accessible in the complete design. Therefore partial classes do not specify access modifiers. The full design of a class that is partial defined may be located at another place within the same diagram or in a different, separate diagram. This technique also allows structuring design diagrams hierarchically by using separate diagrams for full specifications.
In the context of a CoBox class this technique may also be used to hide nested classes inside of the box. This, however, requires that no class outside of this CoBox needs to establish a reference to one of the hidden, nested classes. The principle of hiding unnecessary information in favor of gaining better readability is demonstrated in Figure 4.6.

To further improve readability of design diagrams, CoBoxes that consist only of one class (either due to all nested classes being hidden or simply because there aren’t any) may combine the identical class and box name and display the CoBox semantic by a second, rounded rectangle surrounding the class. An example for this can be seen in Figure 4.7.

**Figure 4.6:** Design simplification 1: Hiding unnecessary information.

**Figure 4.7:** Design simplification 2: Single CoBoxes classes.

**Deployment** Beyond the specification of CoBox classes, their contents and bounds, the specifications also needs to address the deployment of the product line, as this is also modified by features. While UML deployment diagrams generally target this particular problem, the information of those diagrams usually focuses on very technical communication abstractions and provides too much information. CoBoxes are decoupled and independent from each other by nature, therefore, logical deployment is not relevant in this case. Relevant for a product line based on CoBoxes is only physical deployment, the decision which CoBoxes classes should be instantiated on which physical machines. In the context of this work, physical machines creating at least one CoBox are called *deployment targets.*
Figure 4.8: Exemplary deployment specification.

Figure 4.8 depicts an example of the deployment specification. The specification consists of two parts: First, it demonstrates which components are instantiated on which physical machine. Additionally, the diagram shows the number of physical machines the overall system consists of, by specifying the physical connections between deployment targets. The depicted diagram uses only partially specified classes and, therefore, can be seen as a separate deployment diagram. However, including this information in full specifications is also possible.

Example To demonstrate this specification technique, an exemplary configuration of the CoCoME product line is designed. The system allows only cash payment and keyboard input and does not contain any of the optional features that were introduced in the preceding chapter. With this basic configuration, the software system consists of a series of cash desks which are connected to the store server and the corresponding inventory. The resulting design can be seen in Figure 4.9. The CoBox components of the system are directly derived from the system description in Chapter 3. One component represents the cash desk terminal, another one is used for the server. These two CoBoxes run on different physical machines and connect with each other over a central ConnectionAgent component. One is created for each client instance and one for each server instance. The different instances synchronize their state using remote connections and act as a registry for the system. The server and the client instances register at their corresponding instances, which forwards this information to a unique low-level registry. This allows establishing a remote connection between clients and servers.
4 Designing the CoCoME Product Line

Figure 4.9: A sample configuration of the CoCoME product line system.
Each cash desk client uses a set of nested CoBoxes. One for the keyboard, one for the display and one for the printer. As their interfaces are not exposed to outer CoBoxes, they are used for the internal concurrency of the cash desk client only. A client furthermore saves the current order, which consists of a set of order entries. Such an order can be sent to the printer to produce a client’s check. The keyboard can be set into different states to enable and disable certain sets of keys, depending on the state of the payment process. The display allows displaying product information of the last entered product, the total value and the change value at the end of each payment process. These CoBoxes are only specified partially in Figure 4.9. The public methods are displayed to indicate their interfaces.

The store server CoBox also contains two nested CoBoxes, the store inventory and the server GUI. As the inner design of the GUI is not relevant for understanding the general system, it is declared partial and left empty in the figure. Cash desks use the connection agent to establish a reference to the inventory. Using that reference, a cash desk obtains access to product data based on their IDs and adds them to the current order. Once the order is complete, it is sent back to the inventory to update the stock count. Orders and products are modeled as value types, so that they are automatically copied, when sent to other CoBoxes. This allows keeping remote communication at a minimum. Since instances of the inner InventoryEntry class are not referenced from outside their owning CoBox, they do not need a value annotation.

4.3 CoBox based design with variability

The notation introduced above allows creating design diagrams for non-variable CoBox-based software systems. This, however, is not sufficient for covering a software product line and its variability on a design level. To account for this, the following section establishes an extension to the introduced design approach. This extension allows describing the modifications and extensions that have to be applied to a software system in order to integrate a specific functionality into the system. Therefore, designing a software product line is possible by creating a non-variable product line core, using the specification technique above, and a set of feature specifications using the approach described in this section.

Special attention is given to the relation between the feature model and the design of a product line. In existing design notations (as the ones used in [A+02] for example) it is common to create a UML design for the product line and to annotate variabilities of the system with the <<variant>> or similar UML stereotypes. This, however, does not express the reasons for a part of the system being variable and how this variability relates to the product line’s feature model. For automated product derivation, this information is essential, as a configuration of the feature model is automatically translated into an executable software system.

As described in Chapter 2.3.2, altering a generic system artifact requires up to three possible cases. An alteration can include the introduction of new functionality, the removal of existing functionality and modification/adjustment of already present functionality. The general idea for designing alterations in our approach is to attach symbols to deviations from the core design. A + symbol is used for additional class members, − for removed ones and ∗ for modified ones. The symbols + and − in front of an attribute or method name are already used by UML to represent public and private modifiers. Hence, the alterations in the introduced design notation are enclosed by a box and the corresponding symbol is displayed.
in the upper right corner of that box. Additionally, additions are displayed in green, removals in red and modifications to already present functionality are marked in blue.

This can be seen in the example in Figure 4.10. In the example a system core is declared consisting of one CoBox class that uses three nested boxes to implement three public methods. To indicate that the middle section of the figure is a feature specification its name is provided in the hexagon at the top and is called specification name tag. The designed feature adapts CoBox class C by exchanging one of its nested boxes by a different CoBox. Box D is removed, indicated by the class name in red, and box B is added, as shown by the green class name. To reflect all necessary changes two methods of C have to be modified, including its constructor to establish the reference to B. A1 and A2 are not affected by the changes, therefore its specification is declared partial and left empty. Generally speaking, specifications outside of an alteration box have no effect on the core at all, changes have to be marked explicitly. The only reason for including methods, attributes or classes without coloring is to highlight the interface a certain feature uses. With this principle, feature specifications can be kept short, as undisplayed information does not result in unwanted removal of functionality. As a result, method m2 could also be left out of the feature specification without losing any semantical information. Applying the depicted feature specification to the core yields the system design on the right side of Figure 4.10. As specified, CoBox D and method md() are removed, B and method mb() are added. Iteratively applying a set of feature specifications ultimately results in a specific system design for the corresponding product line configuration.

Specifying features requires some additional restrictions and special semantics. These are addressed in the following paragraphs. When more than one feature specification is applied to the core and the corresponding functionality changes (modifying, adding or removing classes, methods or attributes) target the same CoBox, operation priorities apply. Instead of applying the specifications iteratively, their modifications are combined and prioritized by first applying all additions, applying all modifications and finally applying all removals. Assume feature specification A adds a method m to CoBox C and feature specification B removes this method from this CoBox. In case both, A and B, are included in a configuration this priority results in m not being added to C in the configured design. If only B is included
in a configuration, m will not be present and the removal of m is ignored, as it is already not present in the design.

**Specification name tags** The hexagonal name tags are not only for distinguishing feature from core specifications. They can also be used for combining more than one feature identifier, a name for one specific feature of the feature model, with boolean operations. The available boolean operations are negation (!), logical AND (&&), as well as logical OR (||). Each identifier in a name tag is evaluated like a boolean value. Features are TRUE, if they are selected in a configuration, and FALSE otherwise. The specification is applied to the core, if the expression in the name tag evaluates to TRUE. A specification named <F1 && F2> only modifies the core in the described way, if both features are selected in a configuration. In the CoCoME product line this scenario occurs for the credit card payment and express mode features. If a cash desk switches to express mode all non-cash payment options are disabled, including the credit card reader. However, code disabling the device has to be present only if both features are included in a configuration. This code injection is designed by a feature specification with one additional and one modified method and the name tag <CreditCard && ExpressMode>. Only if those two features are included in the configuration, the feature specification is applied to the core.

This syntax also allows negating a boolean value. A specification with the name tag <!F3> modifies the core only, if it is not included in a configuration. This is useful, if a certain feature is usually selected in a configuration and is therefore built into the core, but should still be kept variable. An example for this in the CoCoME product line is the cash payment functionality. This feature is part of the core system. Therefore, in case it is not selected, this functionality has to be removed from the core. This results in a feature specification with name tag <!CashPayment>.

**Deployment** Specifying deployment is different from describing functionality. Deployment information cannot be removed or added. Previous specifications can only be modified. Additionally, deployment information for added CoBoxes may not be present at all, in case these CoBoxes are not nested in a previously existing one. In such a case, the missing information has to be declared.

The requirement for modifying deployment specifications of core CoBoxes can be caused by two cases. Either the deployment of two (or more) targets is combined into one, or the deployment of components on one target is refined into two (or more) separate targets. This is demonstrated in Figure 4.11. Combining two targets from the core specification is shown on the left side of the figure. The specification reads as "Instead of using targets M1 and M2 for deploying the components on the right, they are created on M3". Splitting a deployment target is shown on the right and reads as "Instead of using target M1, the components in the upper right are created on M2 and the components in the lower right are be created on M3".

When specifying the deployment of an additional component, two options are possible. The simplest solution is to state the deployment target in solid lines. This indicates that the component is created on a pre-existing or additional target, no matter how the deployment of previous components is affected by other feature specifications. If the deployment target is not already existing in the core specification, physical connections for the new target have to be introduced, as well. Since a deployment target can be modified by other features, there is no guarantee that its original name is still valid after applying other feature specification.
4 Designing the CoCoME Product Line

Figure 4.11: Example of modifying deployment specification.

For this purpose, the deployment target symbol can be drawn in broken lines. In this case, the name beneath the symbol gives the name of a CoBox class. It expresses that the target of the physical connection is the deployment target instantiating the given CoBox class name.

The broken-lines notation can also be used for directly providing the deployment target of a new component. This is the second notation option, when introducing additional components in a feature specification, and expresses that the new component should be deployed on the same target that is used for deploying the given component. This notation for designing the deployment of a CoBox or the physical connections between two deployment targets is called component-relative deployment.

An example of this notation is shown in Figure 4.12. The core specification is given in the middle of the figure’s left side. It declares a simple client-server scenario, in which an arbitrary number of clients communicates with a single server. Each client deployment target may run several instances of the Client component, whereas the Server component is only created once on a single physical machine. Two feature specifications are shown on the top and bottom of the figure’s left side. The specification of feature X merges the two deployment targets (for the Client and Server components) into one. Feature Y implements new functionality by adding a new server component to the system and modifying the Client component, to allow communication with this new CoBox. The deployment specification of feature Y states that the new CoBox class needs to be instantiated on the deployment target running the Server component.

These two feature specifications can be used in several configurations when deriving a product for this simple product line. Selecting none of the two features, logically yields the unmodified core. Selecting only X and not Y results in the design shown in the figure’s upper right corner, whereas a configuration with feature Y and not feature X yields the design in the lower right corner. Deriving a product with both features included yields the design in the middle of the figure’s right side. This configuration demonstrates the strength of component-relative deployment, as the additional CoBox of feature Y always runs on the same deployment target used for creating the Server component.

4.4 Designing the product line

In this section, the specification technique for (variable) CoBox-based systems are used for designing the CoCoME product line. The first step is finding and designing the core of the product line. This core is designed as a non-variable system and is then made variable by introducing a series of feature specifications.
Figure 4.12: Example of variable deployment specifications.
Usually, the term "core of a product line" is an expression for all shared assets that are not variable in the context of an SPL. In the context of this work, however, the core of a product line describes a basic system that is modified by feature specifications. The difference between these two definitions is that with the usual definition the product line core is not necessarily an actually executable software system. Even if this should be the case, in most cases the SPL core is no valid configuration of the product line. In this work’s context, however, the core of the product line is always defined as an executable system and a valid configuration. This allows creating the core of the product line using regular single application engineering methods.

For the CoCoME product line this means that one of the many configurations of the SPL has to be selected to serve as the product line core. Since adding functionality when selecting a feature usually seems more logical than removing functionality in case it is not selected, the core configuration should be ‘minimal’. Such a minimal configuration can be found using the following steps:

- **Mandatory features**, have to be part of a minimal configuration, as they have to be included in all configurations.

- For **optional features** there is no deterministic way of handling them in the context of a minimal configuration. Usually an optional feature results in additional functionality when selected. In these cases, such features should not be a part of the minimal configuration, which aims at including as little functionality as possible. In some cases, however, when it is very likely for such a feature to be included in almost any configuration, adding the feature to the minimal configuration can still be a valid choice. The core represents a fully functional member of the product line, which can be tested thoroughly without the influence of product line variability. Therefore, including an often used variability might prove beneficial for the overall quality of the product line. An optional feature can also be used to exclude functionality from the system. In analogy to the case presented above, usually the feature should be included in the minimal configuration. This results in a minimal configuration with only few functionality included. However, if the feature is unlikely to be included in future configurations, however, it should be excluded to benefit the testing of the product line core.

In general, including an optional feature in the minimal configuration may have several beneficial or disadvantageous effects on the product line. In the context of the CoCoME product line no clear prediction of which optional features are likely to be selected in future configurations can be made. Therefore the core system does not include any optional features like express support, weighed goods support, etc.

- **Feature selections** in a feature model are used to model a sets of options. Depending on the specific choice at least one or exactly one feature from this set has to be included in a valid configuration. Since the minimal configuration of a product line needs to be a valid configuration a choice has to be made between these options. If the selections allows at least one feature to be selected, for a minimal configuration it should be limited to exactly one feature. The choice of which option to include in the minimal configuration should be based on which of these features is most likely to be included in many configurations.
For the CoCoME product line there are a series of choices that have to be targeted: Payment options, input options and the server platform. For input the keyboard was selected, for payment cash payment was chosen, as these features are most likely included in many configurations. There is no prediction available for the server platform. Windows was chosen as the server platform of the minimal configuration.

In result, the configuration, which serves as the core, is the one used for demonstrating non-variable CoBox diagrams in Figure 4.9. This configuration consists of the features "Cash payment", "Windows server platform", "Multi-desk system" and "Keyboard input". The design of this configuration was already described in Section 4.2. In the following, we focus on the feature specifications extending and modifying the product line core. Of course, this work does not provide enough space for discussing all feature specifications. Instead, three specifications are presented to demonstrate the concepts of the introduced design approach once more. The features "Cash payment", "Single desk system" and "Credit card payment" were chosen, as these features contain all constructs of the introduced design notation. The selected features include the removal of core functionality, modification of deployment and the introduction of additional components.

**Feature "Cash payment"** The specification for the cash payment feature (see Figure 4.13) does not introduce any functionality. As cash payment is already a part of the core system, the specification must only be applied, in case the feature is not part of the configuration. This is expressed by the exclamation mark inverting the value of the feature identifier in
the name tag of the diagram. For removing the cash payment functionality, the keyboard button for initiating cash payment has to be removed by modifying the constructor of the component. This also makes one of the methods of the keyboard obsolete, which is why it is removed from the design. Additionally, the `selectCashPayment` method in the cash desk is also removed. Since the specification does not introduce new functionality, it does not create any additional components and furthermore does not require any deployment specification.

**Feature "Single desk system"** The "Single desk system" feature modifies deployment of the CoCoME system, in case there is no need for client/server separation. Therefore, the deployment targets for the `CashDesk` and `StoreInventory` components can be merged into one. Although the connection agent could be removed for combining client and server applications, it is kept, as other features, e.g. credit card payment (see below), have to access it. Instead, the connection of two local components is altered by modifying the `registerInventory` method of the agent. Additionally, only one instance of the connection agent is created on the "Single client" deployment target. This specification can be seen in Figure 4.14.
Feature "Credit card payment" To include credit card payment in the system a number of modifications on several components are required. After completing an order the cashier presses the "Non-cash payment" button on the keyboard, which activates all available non-cash systems. This includes the credit card reader that is connected to the cash desk. After scanning a card and obtaining the card’s number, the cash desk activates the key pad on the card reader, so that the customer may enter her PIN. To validate the entered security number and that the associated credit account allows paying the price of the order, the cash desk connects to a central bank server. If the payment data was verified, the credit card institute is aware of the payment and the payment process is completed.

To implement this behavior, a new Bank CoBox is added to the system’s design by the specification shown in Figure 4.15. This component is deployed on a bank server deployment target. Establishing a connection between a cash desk and the bank furthermore requires two additional methods in the connection agent. The respective physical connection is designed to connect the bank server with the physical machine running the CashDesk component using component-relative deployment specification.

The CashDesk component itself is extended by adding the card reader to the CoBox, as well as a series of methods. Additionally, the feature specification modifies two methods, namely activateNonCashControllers and deactivateNonCashControllers. These methods are not present in the core specification. However, they are added by the "Non-cash payment" feature, which has to be included in a valid configuration using credit card payment.
4 Designing the CoCoME Product Line

4.5 Summary

In this chapter, we introduced a design notation for specifying CoBox-based software product lines as an extension to UML class diagrams. The notation extends a non-variable CoBox-core by designing a set of transformations adding, removing and modifying functionality and altering deployment information. These transformations are only applied to the core should the current product line configuration match certain criteria given through feature specification name tags. This notation has been used to design the CoCoME product line introduced in Chapter 3. With manual product derivation the resulting design for a specific configuration can be used for a manual implementation. The next step in this work, however, is analyzing how to implement the design using automated product derivation.
5 Implementing the CoCoME Product Line

The last chapter introduced design notations for specifying CoBox-based systems with SPL-oriented variability and resulted in a design for the CoCoME product line. In the following chapter, the implementation of this design and the applied technology for allowing automated product derivation are discussed. Different possibilities for dealing with variability on the implementational level are discussed in Section 2.3.2. The first section analyzes these concepts and selects the one that is most appropriate for implementing the design of the preceding chapter. In succession, the tool implementing the selected concept is introduced. Finally, the implementation of the CoCoME product line and its feature model is described.

5.1 CoBox-based variability management

CoBoxes themselves do not offer any specialized mechanism for dealing with variability, apart from the ones that are offered by usual object-oriented languages. This leaves the choices described in Section 2.3.2 for implementing variability. Dynamic variability management is powerful enough to handle variability in terms of functional behavior. However, the purpose of product lines is to reflect variability only on a dynamic level, if it is explicitly wanted. The static management techniques are therefore of more significance.

Object-orientation Adding functionality using inheritance is undoubtedly possible. Removing functionality is possible in parts at least. The removal of a function, to not be present in the interface anymore, is not possible. However, the execution of a method can be prohibited by adding an empty function with an identical signature in the sub-class. This limitation in removing code also affects the modification of functionality. While adaptation of behavior through code addition works properly in many cases, the removal of code fragments conflicts with the object-oriented paradigm and is therefore not possible. Additionally, creating sub-types is an automated process in almost any object-oriented programming language during inheritance. This creates a problem that would not arise, when modifying the original source code directly. For example: Class C is adapted using inheritance, which results in a new class C'. During adaptation new functionality is added to C', its interface is therefore wider than the C interface. Variable c of type C is used during the execution of method m(). In the adapted system, the variable’s type still is C, the referenced object, however, is of the new type C'. Because of this, the interface of the referenced object appears smaller, than it actually is, effectively masking any extension to the interface made during adaptation. Therefore, the behavior of method m() has to be adapted as well. Constructing the behavior of an object based on the type of a referenced object is not supported in a static manner by the object-oriented paradigm. This problem could be solved in two ways:

- Dynamic resolving of the variability through a runtime type check. Yet, this is not desirable, since variability should be resolved at compile time.
Implementing the CoCoME Product Line

• Adapting behavior of the class, which is accessing the extended interface, through an additional subtype. However, since many languages don’t support inheritance from more than one class, this can create additional problems.

These problems lead to the usual object-oriented concept of inheritance being not powerful enough to handle most product line variability successfully.

Aspect-orientation Using aspect-orientation for implementing product line variability suffers from a problem that object-orientation also has to struggle with: The inability to cleanly remove functionality. Additionally, a case study focusing on aspect-orientation for implementing product line variability [NTW05] finds that “while aspects provide a suitable realization of variability, the aspect-based realization of variability complicates the readability and comprehension of an architecture”. The study concludes that this is caused by “the implicit communication link between aspects and classes”. These disadvantages make aspect-orientation not an ideal choice for implementing the variability of a software product line.

Conditional compilation The configuration of a product line can be expressed as a set of boolean values, as every feature can either be selected or not. Conditional compilation is able to allow quite complex expressions using strings and integers. Boolean values, however, are ”by nature” very basic, which limits the complexity of its expressions. Additionally, code with a lot of overlapping condition statements is exceptionally hard to read. Therefore, this technique should not be the first choice for handling product line variability.

Frames A case study analyzing the benefits of frames over other reuse approaches finds that this technology ”offers significant advantages for implementing product lines over conventional object-oriented techniques” [PM03]. This is mainly to frames fully being able to remove functionality, as well as to add and modify pre-existing functionality. Therefore, this technique is able to fulfill all requirements that were identified for implementational variability management (see Chapter 2.3.2). Although frames can suffer from the same ”implicit communication link” that [NTW05] finds between aspects and classes, the readability of frames is usually still (by far) better, than the results of using conditional compilation or other techniques. Therefore, the CoCoME product line is implemented using frame technology.

5.2 XVCL-based product line variability management

The XML-based Variant Configuration Language (XVCL) [Jar07] is an open-source XML-dialect implementing the originally COBOL-oriented concepts of frame technology [Bas96]. Although it was originally created for a series of Java projects, the tool is decoupled from any specific programming language and allows using the frame concepts for any arbitrary text. This is useful for implementing the CoBox-based CoCoME product line, since JCoBox code is not compatible with regular Java code and using XVCL does not require any customization towards JCoBox.

An introductory XVCL example is shown in Listing 5.1. The example defines several frames, which are used for constructing an exemplary sequence of letters. Everything enclosed by an x-frame tag that is not an XVCL command is considered regular text. Frames D, E and F each contain one break, a text block that may be modified by other frames. Frame A shows a
Listing 5.1: Exemplary set of XVCL frames.
5 Implementing the CoCoME Product Line

Figure 5.2: Resolving the example frame hierarchy. Taken from [XVC09].

Basic example for adapting other frames within a frame. Frames B and C are modified using the adapt command. This modifies a frame and yields the result to frame A, where it replaces the command and continues processing. Since the adapt tag does not contain any child tags to declare specific modifications, frames B and C are processed without any modification caused by superordinate frame A. Frames B and C demonstrate the adaptation of frames with additional modifications. Within the adapt command, a set of insert, insert-after and insert-before commands allows modifying the break points of the frame to be adapted. A frame can be used several times, as shown in frames B and C, which both adapt frame E.

Starting from a top frame, called the specification frame, a set of frames and the adapt commands within define the XVCL frame hierarchy. The hierarchy is a tree with an arbitrary number of children on each node. Each node may be child of an arbitrary number of parent nodes. The frame hierarchy for the above example is depicted in Figure 5.2. The figure also demonstrates the “in-order” traversal through the hierarchy, the results of the processing and which frame is responsible for which part of the end result. Starting from top frame A, processing encounters text ”AAA before” yields it and continues with the adaption of frame B. This frame outputs ”BBB before”, adapts frame D, which yields ”DDD”, outputs ”BBB” and then adapts frame E. In its adapt tag, B specifies that the break tag named ”break.E” is to be modified by inserting the text ”EEE after” after the break. Therefore, processing continues by yielding first ”EEE”, then the inserted text ”EEE after” and finally, returning to frame B and yielding ”BBB after”. After returning to frame A and creating output ”AAA”, the processing of the right half of the frame hierarchy follows an identical route, with the exception that E is modified by inserting text before its break tag.
Implementing the CoCoME Product Line

5.3 Implementing the product line

The CoCoME product line implementation is composed of a set of CoBox components. The corresponding JCoBox source code is structured into a set of XVCL frames. Every component’s JCoBox code is created from a set of component-specific XVCL frames. For each component that is part of the system core, a core frame is created from the core specification of the system. Except for a set of break point tags, core frames consist of regular JCoBox code and removing all XVCL tags results in an executable software system according to the core design specification (see Figure 4.9). These core frames are extended and modified by feature frames. Each feature frame targets exactly one component’s core frame and implements the changes specified by exactly one feature specification for this particular component. This means that a feature specification as described in Section 4.3, which introduces CoBox C and modifies CoBox A and B, results in exactly three feature frames, one for each A, B and C. This allows creating a set of frames that is derived directly from the product line’s core and feature specifications. Feature specifications are simply divided into one feature frame for each affected component. The actual modifications designed in the product line’s feature specifications are implemented by a set of insert, insert-before and insert-after tags in the corresponding feature frames.

The design of the feature frames and the frame hierarchy, is not obvious and requires additional discussion. It seems logical to design a frame hierarchy with a single specification frame sequentially adapting all components and writing the result to a series of output files. In this scenario, with n components (potentially) in the system, the top frame has n child frames, of which each is configured by sequentially adapting the core frame through a set of feature frames. This, however, is not possible in XVCL, at least not generally speaking. The example in Listing 5.3 demonstrates this problem. In the listing, frame C represents the core frame, A and B are both feature frames. Both feature frames modify C by inserting code before the ”break_C” break. It seems logical to expect the result ”AAA BBB CCC”. This, however, is not the actual result. In XVCL adaptations of superordinate frames replace

Listing 5.3: Product line-ineligible XVCL-behavior of subsequent adapt commands.

```
Listing 5.3: Product line-ineligible XVCL-behavior of subsequent adapt commands.

5.3 Implementing the product line

The CoCoME product line implementation is composed of a set of CoBox components. The corresponding JCoBox source code is structured into a set of XVCL frames. Every component’s JCoBox code is created from a set of component-specific XVCL frames. For each component that is part of the system core, a core frame is created from the core specification of the system. Except for a set of break point tags, core frames consist of regular JCoBox code and removing all XVCL tags results in an executable software system according to the core design specification (see Figure 4.9). These core frames are extended and modified by feature frames. Each feature frame targets exactly one component’s core frame and implements the changes specified by exactly one feature specification for this particular component. This means that a feature specification as described in Section 4.3, which introduces CoBox C and modifies CoBox A and B, results in exactly three feature frames, one for each A, B and C. This allows creating a set of frames that is derived directly from the product line’s core and feature specifications. Feature specifications are simply divided into one feature frame for each affected component. The actual modifications designed in the product line’s feature specifications are implemented by a set of insert, insert-before and insert-after tags in the corresponding feature frames.

The design of the feature frames and the frame hierarchy, is not obvious and requires additional discussion. It seems logical to design a frame hierarchy with a single specification frame sequentially adapting all components and writing the result to a series of output files. In this scenario, with n components (potentially) in the system, the top frame has n child frames, of which each is configured by sequentially adapting the core frame through a set of feature frames. This, however, is not possible in XVCL, at least not generally speaking. The example in Listing 5.3 demonstrates this problem. In the listing, frame C represents the core frame, A and B are both feature frames. Both feature frames modify C by inserting code before the ”break_C” break. It seems logical to expect the result ”AAA BBB CCC”. This, however, is not the actual result. In XVCL adaptations of superordinate frames replace
adaptations in subordinate frames. In the above example, this means that the insert-before tag of frame A overwrites the one in frame B, before it is used for finally adapting the generic frame C. Therefore, processing the above structure results in “AAA CCC”.

This behavior usually is beneficial for more complex scenarios. Consider frame C holding the generic implementation of a class that is used in two projects P1 and P2. These two projects share some requirements targeting C, but are not identical towards the configuration of the generic implementation. To modify C according to the exact needs of P1, frame B adapts C and specifies a wide set of modifications. For tailoring C to the requirements of P2, frame A is used. Since P1 and P2 share similarities, some modifications of B could be copied over to A for adapting the generic frame. To improve reuse, however, A can also adapt B and overwrite modifications of B that are not required for project P2. A and B can then be processed separately for obtaining different specialized versions of C. Processing frame A yields the specialized version for P2, while processing frame B results in a specialized version for P1. This example demonstrates that for the general construction of code this behavior is useful. For implementing the CoCoME product line, with its core and feature specifications, however, this behavior is not desirable.

Figure 5.4: Automated product derivation for the CoCoME product line using XVCL.
Instead of constructing a single frame hierarchy, the product line implementation uses a two-step configuration process. In a first step, the *accumulation step*, all modifications for a specific component under the given configuration are merged into a single frame by concatenating the *insert* commands of several feature frames. A set of alterations targeting the same break point within one frame don’t overwrite each other. Instead, XVCL automatically combines several *insert* tags targeting the same break point. Therefore, it is sufficient to create one frame with a single *adapt* tag and a concatenation of all selected alteration tags. The frame holding the combined modifications of several feature frames for one system component is called its *configuration frame*. It is a temporary result for one specific configuration process. In the example discussed above two feature frames A and B targeting an identical breakpoint with two *insert* statement resulted in one modification being overwritten, due to hierarchical structuring. Accumulating both feature frames into one configuration frame results in a frame consisting of a single *adapt* statement with two subordinate *insert* tags. These are automatically combined by XVCL when processing the frame and comply with the operational priority rule established in Chapter 4.3.

In the second step of the process, called *configuration step*, the previously created configuration frames are used to adapt the corresponding components’ core frames. Given that the feature frames accumulated in the first phase are the correct frames for a given configuration, this process results in correctly configured components and, therefore, in a correctly configured system. The process is depicted in Figure 5.4. Once the accumulation and configuration steps are completed, the resulting JCoBox code is compiled into Java code and sent through a regular Java compiler.

To ensure that only the features that are appropriate for a given configuration are selected in the accumulation phase, two issues have to be addressed. First, the current configuration has to be defined for the XVCL process and, second, based on this configuration, a feature selection mechanism has to be established. To define the current configuration, XVCL allows declaring a set of variables. Each variable corresponds to one feature of the feature model and may be defined either to "yes" or "no". This can be seen in Listing 5.5. The XVCL *set* command is used to declare the features of the CoCoME feature model. The name of

```
<x-frame name="cocome_configuration">
  <!-- Payment features, choose at least one -->
  <set var="FEATUREPAYMENTCASH" value="yes"/>
  <set var="FEATUREPAYMENTNONCASH" value="yes"/>

  <!-- Non-cash payment features -->
  <set var="FEATUREPAYMENTCREDITCARD" value="yes"/>
  <set var="FEATUREPAYMENTPREPAIDCARD" value="yes"/>

  <!-- Optional system features -->
  <set var="FEATURESINGLEDENESYSTEM" set to "no"/>
  <set var="FEATUREPAYMENTNONCASH" set to "yes"/>
  <set var="FEATUREINPUTSCANNER" set to "yes"/>
  <set var="FEATURESELFSERVICESUPPORT" value="no"/>

  ...
</x-frame>

Listing 5.5: Configuration file of the CoCoME product line used in XVCL-based automated product derivation.
```
5 Implementing the CoCoME Product Line

these features is given by the \texttt{var} property, the value of the feature, indicating if it is part of the configuration, is given by the \texttt{value} property of the tag. The provided example defines a configuration which includes credit card, prepaid card and cash payment options and does not include self service support. In the current configuration this declaration is handled manually, meaning a developer has to modify this declaration based on the configuration she wants to derive. However, a GUI for this feature selection can be easily created.

To select only features that are appropriate for the declared configuration, these variable declarations have to be checked in the accumulation phase, before including a specific feature frame in a configuration frame. This is achieved by using \textit{build frames}. A build frame is created for every component requiring configuration and captures the information for selecting its feature frames. This information corresponds to the specification name tags of all feature specifications affecting a particular system component. For the CoCoME product line implementation we created these build frames manually. However, they were created systematically from the feature specifications of the product line, without further creative input, so that this task can be automated with proper tool support.

To ensure a valid and executable software system, the declared configuration has to be valid according to the product line’s feature model. Therefore, the accumulation step will check the feature model’s constraints before starting with the actual accumulation process. The constraints of the feature model are included in a constraint frame, which is presented in Appendix A.1. Should the declared configuration be invalid, the accumulation step is aborted and no results are produced.

To demonstrate the configuration process in further detail, in the following we analyze one exemplary component, the CashDesk component. Listing 5.6 contains an excerpt of the CashDesk’s build frame, which is used in the accumulation phase. In analogy to the specification name tags of the “cash payment” and “credit card payment” features depicted in Figures 4.13 and 4.15, the build frame examines the currently defined configuration and includes the modifications of the respective feature frames, in case the name tag’s conditions are met. For the credit card payment feature frame this means that it is only included in the accumulation phase, should the \texttt{FEATURE\_PAYMENT\_CREDITCARD} variable be defined to "yes".

\begin{verbatim}
<x-frame name="CashDesk\_BUILD" outfile="CashDesk\_CONFIGURATION\_frame">
  ...
  <select option="FEATURE\_PAYMENT\_CREDITCARD">
    <option value="yes">
      <adapt x-frame="CashDesk/CashDesk\_FEATURE\_PAYMENT\_CREDITCARD\_frame" src="yes"/>
    </option>
  </select>
  ...
</x-frame>
\end{verbatim}

\textbf{Listing 5.6: XVCL build frame for the CashDesk component.}
5 Implementing the CoCoME Product Line

The cash payment feature frame, however, is only included during accumulation, should the FEATURE_PAYMENT_CASH variable be defined to "no". This is required, as the functionality for cash payment is part of the system core. The feature frame removes this functionality according to the specification shown in Figure 4.13.

One of the CashDesk’s feature frames is shown in Listing 5.7. Based on the corresponding feature specification (see Figure 4.15), the frame expands the CashDesk component to allow credit card payment. This requires an additional CardReader attribute, its creation in the constructor, enabling and disabling it at various occasions and dealing with the credit card payment process itself. If the conditions in the build frame for including this feature frame are fulfilled, the insert statements, along with modifications of other feature frames, are assembled in the CashDesk’s configuration frame. As declared in the build frame this temporary file is named "CashDesk.CONFIGURATION.frame". This file is included in Appendix A.2, under the configuration defined in Listing 5.5. This configuration includes all available payment options, as well as scanner input support. Combining this configuration frame with the CashDesk core frame results in a configured version of the CashDesk JCoBox code, according to the product line’s design. After configuration the resulting JCoBox code is compiled into Java code, which is compiled into executable Java byte code.

Listing 5.7: CashDesk feature frame for the "Credit card payment" feature according to the specification in Figure 4.15.
Performing the accumulation and configuration steps for all system components results in a correctly configured member of the product line matching its corresponding core and feature specifications. This derivation process is fully automated. In the current implementation all steps of the derivation are integrated into an Eclipse project. Modifying the configuration declaration or any source code file, automatically results in an executable product according to the declared configuration. Depending on the configuration, this ultimately results in the cash desk application, the store server and the enterprise server, a subset of these applications or one single desk application, combining the functionality of such a subset. A screenshot of the cash desk application is shown in Figure 5.8.

5.4 Summary

In this chapter we used an XVCL-based approach to implement the CoBox-based CoCoME product line design explained in Chapter 4. Deriving an executable software product requires a series of steps, which are all fully automated. Due to this automation, changing the configuration automatically produces new Java source code, reflecting these changes. This approach can be used for an arbitrary software product line based on the design notation from Chapter 4 and results in a product line implementation capable of automated product derivation. This automated product derivation is also relatively fast. Configuring the JCoBox code of the CoCoME product line takes about 2 seconds on a Centrino notebook. Compiling the code to Java and to Java byte code afterwards requires an additional 4 seconds. This demonstrates that the configuration phase is also reasonably fast.
Figure 5.8: Screenshot of the configured and compiled CashDesk application, including credit card, prepaid card and cash payment, as well as both keyboard and scanner input.
6 Formal Analysis of Software Product Lines

The previous chapters introduced the tasks for modeling, designing and implementing the CoCoME product line. Since the functional domain of the example is not relevant for any of the introduced concepts, these tasks allow creating generic software product lines with fully automated product derivation. In a real-world scenario, this allows a software development organization to tailor a configuration of a software product line to the desires of a customer in a very simple way. The desired features are selected, and automated product derivation creates the fully configured software product. However, in complex real-world software product lines the derivation of a product often includes introducing additional functionality previously not offered by the product line, or specialized optimizations, for example to match specific non-functional constrains.

This chapter formalizes the experience from developing the CoCoME product line to find a general process for constructing a product line using automated product derivation, based on the introduced concepts. This approach is explained in two steps. First, the initial product line is created from a set of features, in analogy to the efforts presented in Chapters 3, 4 and 5. Second, after the initial creation the resulting product line can be adapted and modified during its life cycle based on new features and optimizations. In both phases, the requirements for applying the introduced approach are formally captured and resulting properties of the product line are analyzed.

6.1 Initial product line creation

Model and requirements phase In the proposed product line approach, the first step is to initially create the variability of the software product line. In analogy to the exemplary CoCoME product line, a feature model has to be created. A feature model defines a set of features that are relevant for the product line and identifies valid configurations for the SPL.

Definition 6.1 (Feature model)

A feature model FM is a tuple $FM := (F, fv)$, where $F$ is a non-empty, finite feature set $F := \{F_1, F_2, ..., F_m\}$, $m \geq 1$ and $fv$ is a validation function, which identifies subsets of the feature set as valid configurations.

$$fv : C \mapsto \begin{cases} true & \text{if } C \subseteq F \text{ and } C \text{ is valid configuration} \\ false & \text{otherwise} \end{cases}$$

If $fv(C)$ yields true for a configuration $C$, this configuration is called a valid configuration. Otherwise, $C$ is an invalid configuration. The feature model is a result of the initial analysis, requirements and modeling phase.

Definition 6.2 (Set of valid configurations)

Let $FM = (F, fv)$ be a feature model. The set of valid configurations for $FM$, is defined by $valid(FM) := \{C \mid fv(C) = true\}$
According to the definition of $fv$, all elements in $valid(FM)$ are subsets of $F$. As an additional property, a feature model should define only features that can be used to express variability. This results in the definition of minimal feature models.

**Definition 6.3 (Minimal feature model)**

A feature model $FM = (F, fv)$ is called minimal, iff its feature set does not contain any features, which are irrelevant for the validity of a configuration.

$$FM \text{ is minimal } \iff \nexists F_i \in F, \forall C : fv(C) = fv(C \setminus \{F_i\}) = false$$

The definition can also be re-phrased to the condition that $F$ must not contain a feature that is not present in any valid configuration:

$$FM = (F, fv) \text{ is minimal } \iff \nexists F_i \in F, \forall C \in valid(FM) : F_i \notin C.$$

Minimal feature models seem to be rather theoretical constructs. To demonstrate that the definition is useful and that it is actually possible to design non-minimal feature models, a short example feature diagram is considered:

The feature model described by the diagram defines a feature set $F = \{A, B, C\}$. Based on the constraints expressed in the diagram the set of valid configurations consists of the configurations $C_1 = \{C\}$ and $C_2 = \{A, C\}$. A configuration including feature B is not valid, as it requires feature A, which excludes B. As B matches $\forall C \in valid(FM) : B \notin C$, the feature model is not minimal. Minimal feature models can be compared based on their valid configurations.

**Lemma 6.4 (Equality of feature models)**

Two minimal feature models $FM_A$ and $FM_B$ are identical, iff their corresponding set of valid configurations are identical.

$$FM_A = FM_B \iff valid(FM_A) = valid(FM_B)$$

**Proof:** Let $FM_A = (F_A, fv_A)$ and $FM_B = (F_B, fv_B)$ be minimal feature models.

$\Rightarrow$ If $F_A$ and $F_B$ are identical, their feature sets and validation functions are equal. Therefore:

$$valid(FM_A) = \{C \subseteq F_A \mid fv_A(C) = true\} = \{C \subseteq F_B \mid fv_B(C) = true\} = valid(FM_B)$$

$\Leftarrow$ Proof by contradiction. Let $valid(FM_A) = valid(FM_B)$ and $FM_A \neq FM_B$ hold. According to the definition of a feature model this means that at least one of $fv_A \neq fv_B$ and $F_A \neq F_B$ holds.

- $fv_A \neq fv_B \rightarrow valid(FM_A) \neq valid(FM_B) \rightarrow$ contradiction
- $F_A \neq F_B$ (w.l.o.g. $|F_A| > |F_B|$)

If $valid(FM_A) = valid(FM_B)$ holds then $F_B$ must be a subset of $F_A$ and the additional features of $F_A$ are not relevant for the validity of a configuration.

$\Rightarrow \exists F_i \in F_A, \forall C \subseteq F_A : fv_A(C) = fv_A(C \setminus \{F_i\}) = false \rightarrow$ Contradiction to $FM_A$ minimal.
Design phase In the design phase, the feature model is used to construct the design of the product line. Of course, the design process is a creative and unpredictable task. Even if creating an invariable single software system serving as the core of a product line was a non-creative task, the choice between several possible minimal configurations still introduces numerous possible product line designs. The result of this creative design task $\delta$ is a single core specification and a set of feature specifications. Feature specifications correspond to the feature diagrams introduced in Chapter 4.3 and consist of a transformation function and an indicator function. The transformation function of such a specification captures the added, removed and modified functionality of a diagram. The diagram’s name tag is represented by the specification’s indicator function.

**Definition 6.5 (Product line specification)**

The specification of a product line is defined by a tuple $SPLSpec := (CSpec, FSpec)$ consisting of a core specification $CSpec$ and a set of feature specifications $FSpec$ where

- $FSpec$ is a set of feature specifications
  
  $FSpec := \{FS_1, FS_2, ..., FS_n\}$

- A feature specification is a tuple consisting of an indicator function $FI$ and a transformation function $FT$.
  
  $\forall i \in [1..n] : FS_i := (FI_i, FT_i)$

- The transformation function of a feature specification modifies the core specification $CSpec$ and yields a new core specification $CSpec'$.
  
  $FT_i : CSpec \mapsto CSpec'$

- The indicator function of a feature specification indicates if the transformation function should be applied to the core under a given configuration.
  
  $\forall i \in [1..n] : FI_i : C \subseteq F \mapsto \begin{cases} true, & \text{if feature is applied in configuration } C \\ false, & \text{otherwise} \end{cases}$

- Each transformation function contains removals, modifications or additions only and the indexes are sorted, such that
  
  $FT_1 \ldots FT_j$ contain removals only,
  
  $FT_{j+1} \ldots FT_k$ contain modifications only and
  
  $FT_{k+1} \ldots FT_n$ contain additions only.

Additionally, it is required that

- $SPLSpec$ is based on a feature model $FM = (F, fv)$ and is derived during specification by the creative design task $\delta$
  
  $SPLSpec = \delta(FM)$

  and that

  - one valid configuration must not result in any feature transformation
  
    $\exists C \in valid(FM), \forall i \in [1..n] : FI_i(C) = false$
In analogy to the name tags of feature diagrams, the result of the indicator function does not necessarily correspond to a single feature of the feature set. Instead, as for the boolean expression in the diagram name tag, the result may be defined based on a set of features connected by Boolean operations. The one valid configuration that must not result in any feature transformation is the minimal configuration that forms the product line core, as explained in Chapter 4.4. The separation of additions, removals and modifications can be achieved by splitting every specification not complying with this constraint into up to three separate specifications. This separation is important to assure that the application of transformations in a sequential order conforms to the operation priority rule in Chapter 4.3.

Based on a configuration of the feature model, the core and feature specifications can be used for obtaining a configured specification, the specification for a particular configuration. This is done by iteratively applying to the core all feature transformation functions, of which the corresponding feature indicator function evaluates to true under the current configuration. This is shown in algorithm 1. Declaratively this configuration can also be defined as follows.

**Definition 6.6 (Configuration function)**

Let \( FM \) be a feature model, \( SPLSpec = (CSpec, FSpec) = \delta(FM) \) the resulting product line specification and \( C \in valid(FM) \) a valid configuration. The configured specification for configuration \( C \) is defined by

\[
\text{Configure}(C, CSpec, FSpec) := \begin{cases} 
(CSpec, \emptyset), & \text{if } FSpec = \emptyset \\
\text{Configure}(C, FT_n(CSpec), FSpec\{FS_n\}), & \text{if } FS = \{FS_1, \ldots, FS_n\} \text{ and } FI_n(C) = true \\
\text{Configure}(C, CSpec, FSpec\{FS_n\}), & \text{otherwise}
\end{cases}
\]

Applying feature transformations with higher indexes first and the sorting established by Definition 6.5 guarantees that the result of the configuration process satisfies the operation priority rule explained in Chapter 4.3. The transformations that are applied to the core specification under a given configuration are captured by the following definition.

**Definition 6.7 (Set of applied features)**

Let \( SPLSpec = (CSpec, FSpec) \) be a product line specification and \( C \) a configuration. The set of applied features for \( SPLSpec \) and \( C \), is defined by

\[
\text{applied}(SPLSpec, C) := \{ F = (FI, FT) \in FSpec \mid FI(C) = true \}.
\]

**Algorithm 1: Configuring a specification**

```plaintext
Input: CSpec, FSpec, C
repeat
  Select FS_n from FSpec = \{FS_1 \ldots FS_n\}
  FI_n(C) = true \Rightarrow CSpec := FT_n(CSpec)
  FSpec := FSpec\{FS_n\}
until FSpec = \emptyset
Result: CSpec, \emptyset
```
The definitions of the \textit{Configure} function on product line specifications allow deriving a configured specification. However, the original definition also captures specifications with too little variability. A specification with feature specifications that are never applied (because their indicator function does not evaluate to \textit{true} for any valid configuration), for example, yields the same configured specification for all configurations. Clearly, this is not useful when discussing product lines, as configurations are supposed to represent system variability. This example is a special case of a more general scenario, in which more than one valid configuration results in the same set of applied features. Therefore, the following definition sharpens the term of product line specifications by additional properties:

\textbf{Definition 6.8 (Well-formed specifications)}

Let $SPLSpec = (CSpec, FSpec) = \delta(FM)$, with feature model $FM = (F, fv)$, be a product line specification according to Definition 6.5. $SPLSpec$, is well-formed, iff the following conditions hold:

i) No feature transformation of $SPLSpec$ results in an identity function for the core specification.
$$FT_n(CSpec) = CSpec' \neq CSpec$$

ii) No valid configuration contains a set of features that builds an identity function for the core specification.
$$\forall C \in \text{valid}(FM), \#F_{i_1}, \ldots, F_{i_n} \in FSpec, \forall j \in [1..n] : FI_{i_j} = \text{true} \land FT_{i_j}(\ldots(FT_{i_n}(CSpec))) = CSpec$$

iii) All feature indicator functions evaluate to \textit{true} for at least one valid configuration.
$$\forall (FI_i, FT_i) \in FSpec, \exists C \in \text{valid}(FM) : FI_i(C) = \text{true}$$

iv) Two feature specifications must not result in the same transformations.
$$\#F_1 = (FI_1, FT_1), F_2 = (FI_2, FT_2) \in FSpec : F_1 \neq F_2 \land FT_1 = FT_2$$

v) All valid configurations of the feature model must result in a different set of applied features. Let $C, C' \in \text{valid}(FM)$ be two valid configurations. Then
$$\text{applied}(SPLSpec, C) = \text{applied}(SPLSpec, C') \Leftrightarrow C = C'$$

The definition of a well-formed specification ensures that a specification is useful and that the configurations of the feature model capture actual variability in the specification. Non-well-formed specifications comprise feature transformations which are either never applied or redundant. A specification that does not fulfill criterion i) of Definition 6.8, for example, contains a feature transformation which does not actually provide any new or modified functionality, as the core is not modified at all. The properties of well-formed specifications allow formulating the following lemma.

\textbf{Lemma 6.9 (Equality of configured specifications)}

Let $SPLSpec = (CSpec, FSpec) = \delta(FM)$ be a well-formed product line specification. Then it holds that two configured specifications, $\text{Configure}(CSpec, FSpec, C)$ and $\text{Configure}(CSpec, FSpec, C')$ are identical, iff the configurations $C$ and $C'$ are identical.
$$\text{Configure}(CSpec, FSpec, C) = \text{Configure}(CSpec, FSpec, C') \Leftrightarrow C = C'$$
Proof:

\( \Leftarrow C = C' \) leads to \( \text{Configure}(C_{\text{Spec}}, F_{\text{Spec}}, C) = \text{Configure}(C_{\text{Spec}}, F_{\text{Spec}}, C') \).

\( \Rightarrow \) The configured specifications are identical and the end results are both based on the same core specification. Therefore, the transformations that lead to the configured specifications are equal. Since a feature transformation of a well-formed specification cannot be empty (6.8 i)) or reversible (6.8 ii)) and no two transformation yield an identical transformation (6.8 iv)), the applied transformations must be identical. According to Definition 6.8 v), this yields \( C = C' \).

The term minimal configuration, introduced in Chapter 4, to describe the configuration with minimal variable functionality, can also be described using the definitions above. It is the configuration that yields the unmodified core specification as its configured specification.

Definition 6.10 (Minimal configuration)

A configuration \( C \) that does not modify the core of a specification is called a minimal configuration.

\[ C \text{ is minimal} \iff \text{Configure}(C_{\text{Spec}}, F_{\text{Spec}}, C) = C_{\text{Spec}} \]

Furthermore, in a well-formed specification, there is exactly one minimal configuration.

Lemma 6.11 (Minimal configuration of a well-formed specification)

Let \( C \) be a minimal configuration for a product line specification \( SPL_{\text{Spec}} = (C_{\text{Spec}}, F_{\text{Spec}}) \). If \( SPL_{\text{Spec}} \) is well-formed, then it holds that \( C \) is unique.

Proof: According to Definition 6.5 in every product line specification one valid configuration has to exist, for which no feature transformations are applied to the core. This configuration is obviously minimal.

Let \( C^* \) be that minimal configuration. Since \( \text{Configure}(C_{\text{Spec}}, F_{\text{Spec}}, C^*) = C_{\text{Spec}} = \text{Configure}(C_{\text{Spec}}, F_{\text{Spec}}, C) \) holds and \( SPL_{\text{Spec}} \) is well-formed, \( C \) and \( C^* \) must be identical (Lemma 6.9), which and yields the result.

Implementation phase For the implementation of the CoCoME product line specification (see Chapter 5), the core specification is refined into a set of core frames, one for each CoBox. Feature specifications, furthermore, result in a feature frame for each CoBox that is added, removed or modified by the feature specification. Each feature frame modifies the source code of its corresponding core frame, if the feature frame is included in a given configuration. In other words: A feature frame transforms its corresponding core frame, if the indicator of the frame’s corresponding feature specification selects the transformation to be applied. Therefore, the core frame of a component (i.e. CoBox in the example) and the feature frames of that component can be seen as a product line specification (according to Definition 6.5) for one component, with the difference that we consider code instead of specification diagrams. As a result, the implementation of a product line can be seen as a set of component specifications, such that the following definition arises:

Definition 6.12 (Product line implementation)

The implementation of a product line is defined by a tuple \( SPL_{\text{Impl}} := (C_{\text{Impl}}, F_{\text{Impl}}) \) consisting of a set of core frames \( C_{\text{Impl}} \) and a set of feature frames \( F_{\text{Impl}} \) where
• $CImpl$ is a set of core frames, called the core implementation

\[
CImpl := \{CF_1, \ldots, CF_n\}
\]

• $FImpl$ is a set of feature frames

\[
FImpl := \{FF_1, FF_2, \ldots, FF_m\}
\]

• A feature frame $FF_i$ is a tuple consisting of an indicator function $FFI_i$ and a transformation function $FFT_i$.

\[
\forall i \in [1..m] : FF_i := (FFI_i, FFT_i)
\]

• The transformation function of a feature frame modifies the core frames $CImpl$ and yields new core frames $CImpl'$. 

\[
FFT_n : CImpl \rightarrow CImpl'
\]

• The indicator function of a feature frame indicates if the transformation function should be applied to the core under a given configuration.

\[
\forall i \in [1..m] : FFI_i : C \subseteq F \mapsto \begin{cases} true, & \text{if feature frame is applied in configuration C} \\ false, & \text{otherwise} \end{cases}
\]

• Each transformation function contains removals, modifications or additions only and the indexes are sorted, such that $FFT_1 \ldots FFT_i$ contain removals only, $FFT_{i+1} \ldots FFT_j$ contain modifications only and $FFT_{j+1} \ldots FFT_n$ contain additions only.

• Each feature frame either introduces a new component to the core or modifies exactly one component of the core.

\[
\forall i \in [1..m] : FFT_i(CImpl) = CImpl \cup \{CF_{n+1}\} \\
\forall FFT_i(CImpl) = \{CF_1, \ldots, CF_{i-1}, CF_i', CF_{i+1}, \ldots, CF_n, \}
\]

Additionally, we require that

• $SPLImpl$ is the result of an implementation function $\gamma$ and is built from the product line specification that is derived from a feature model.

\[
SPLImpl = \gamma((CSpec, FSpec)) = \gamma(\delta(FM))
\]

• The feature indicator functions of the implementation are consistent with the ones of the specification.

\[
\forall (FFI_i, FFT_i) \in FImpl, \exists (FI_j, FT_j) \in FSpec : FFI_i = FI_j \\
\forall (FI_j, FT_j) \in FSpec, \exists (FFI_i, FFT_i) \in FImpl : FFI_i = FI_j
\]

The number of core frames contained in $CImpl$ is given by the number of components that are defined in the product line’s core specification. The number of feature frames cannot be determined in general, as a feature specification may contain an arbitrary number of
additional components which results in an arbitrary number of feature frames. As described in Chapter 5, each feature specification in the design phase is split into a number of feature frames, such that each feature frame modifies exactly one core frame or adds a new component to the core. Apart from this additional condition, the implementation of a product line is structurally identical to a product line specification according to Definition 6.5.

In contrast to Definition 6.7, in a product line implementation there are no applied features, but applied feature frames.

**Definition 6.13 (Set of applied feature frames)**

Let $SPL_{Impl} = (C_{Impl}, F_{Impl})$ be a product line implementation and $C$ a configuration. The set of applied feature frames for $SPL_{Impl}$ and $C$, is defined by

$$\text{applied}(SPL_{Impl}, C) := \{ F = (FFI_i, FFT_i) \in F_{Impl} \mid FFI_i(C) = true \}.$$  

In analogy to Definition 6.8, a well-formed product line implementation ensures that it appropriately captures the variability of the corresponding feature model.

**Definition 6.14 (Well-formed implementation)**

Let $SPL_{Impl} = (C_{Impl}, F_{Impl}) = \gamma(\delta(FM))$, with $FM = (F, fv)$, be a product line implementation according to Definition 6.12. $SPL_{Impl}$, is well-formed, iff the following conditions hold:

i) No valid configuration contains a set of features that builds an identity function for the core implementation.

$$\forall C \in \text{valid}(FM), \exists FFI_{i1}, \ldots, FFI_{in} \in F_{Impl}, \forall j \in [1..n] : FFI_{ij} = true \land FFT_{i1}(\ldots(FFT_{in}(C_{Impl}))) = C_{Impl}$$

ii) All feature indicator functions evaluate to true for at least one valid configuration.

$$\forall (FFI_i, FFT_i) \in F_{Impl}, \exists C \in \text{valid}(FM) : FFI_i(C) = true$$

iii) Two features must not result in the same transformations.

$$\exists FFI_1 = (FFI_{i1}, FFT_{i1}), FFI_2 = (FFI_{i2}, FFT_{i2}) \in F_{Impl} : FFI_1 \neq FFI_2 \land FFT_1 = FFT_2$$

iv) All valid configurations of the feature model must result in a different set of applied feature frames. Let $C, C' \in \text{valid}(FM)$ be two valid configurations.

$$\text{applied}(SPL_{Impl}, C) = \text{applied}(SPL_{Impl}, C') \Leftrightarrow C = C'$$

Two additional facts are worth noting when discussing a frame-based product line implementation:

- There are no 'empty' frames without any actual modification. Definition 6.12 ensures, that each feature frames modifies exactly one component or introduces a new one. Therefore, condition i) in the well-formedness definition of product line specifications (6.8) does not have a corresponding counterpart in the definition of a well-formed implementation above.

- Frame conditions can be structured, so that no frame removes functionality that is added by another frame. Example: Frame A adds functionality to component C if expression
6 Formal Analysis of Software Product Lines

$C_A$ evaluates to true and frame B removes that functionality from component C in case expression $C_B$ evaluates to true. Rearranging the condition of A to '$C_A \land \neg C_B$' and deleting B results in the same behavior when applying feature frames.

These properties, combined with the structural similarities of product line specifications and implementations, suggest that the well-formedness of a product line specification can be propagated to its implementation. Unfortunately, the fact that a specification is well-formed alone does not guarantee that the resulting implementation is well-formed. However, the reverse holds, given that the implementation was created 'reasonably' from its specification, by transforming each specification into a corresponding set of frames as explained in Chapter 5. Capturing this 'reasonable' implementation is beyond the scope of this work, because it requires the following three challenging steps:

a) Introducing formal semantics to product line specifications.

b) Ensuring that an implementation is created from a product line specification by dividing this specification into a set of single-component specifications.

c) Validating that a set of frames implements these component-oriented specifications, such that they are semantically consistent with the product line specification.

A formalization of this semantical correlation between a product line implementation and its specification would allow formulating a lemma, instead of an informal remark.

**Remark 6.15 (Well-formed product line implementation)**

Let $SPLSpec = (CSpec, FSpec)$ be a product line specification and $SPLImpl = (CImpl, FImpl) = \gamma(SPLSpec)$ the corresponding implementation. If $SPLImpl$ is well-formed and the implementation was created 'reasonably' from $SPLSpec$, then it holds that $SPLSpec$ is well-formed.

**Proof:** 'Proof' by contradiction. Let $SPLImpl$ be well-formed and $SPLSpec$ not well-formed. Then at least one of conditions i) - v) in Definition 6.8 does not hold.

i) $SPLSpec$ contains a feature transformation that is an identity function for the core specification. This results in a set of feature frames in the implementation that are an identity for the core frames. $\Rightarrow$ $SPLImpl$ is no product line implementation according to Definition 6.12 $\Rightarrow$ Contradiction

ii) $SPLSpec$ contains a set of feature transformations that are an identity function for the core specification. If this holds for the whole specification, it holds especially for one component. $\Rightarrow$ $SPLImpl$ not well-formed $\Rightarrow$ Contradiction

iii) $SPLSpec$ contains a feature indicator that evaluates to false for all valid configurations. Based on Definition 6.12 this indicator is also present in $SPLImpl$. $\Rightarrow$ $SPLImpl$ not well-formed $\Rightarrow$ Contradiction

iv) $SPLSpec$ contains two identical feature transformations. Since a specification is split along the bounds of the system components into a set of feature frames, this especially means that at least one component in $SPLImpl$ is modified identically by two different feature frames. $\Rightarrow$ $SPLImpl$ not well-formed $\Rightarrow$ Contradiction

v) In $SPLSpec$, two valid non-identical configurations result in the same set of applied transformations. Therefore, especially one component is modified identically under both configurations. $\Rightarrow$ $SPLImpl$ not well-formed $\Rightarrow$ Contradiction
Because of the structural similarities between product line specifications and implementations, configuring the implementation of a product line is very similar to configuring the specification it was created from.

**Definition 6.16 (Configuration of SPL implementations)**

Let $FM$ be a feature model, $SPLImpl = (CImpl, FImpl) = \gamma(\delta(FM))$ the resulting product line implementation and $C \in valid(FM)$ a valid configuration.

The configured implementation for configuration $C$ is defined by

\[
Configure(C, CImpl, FImpl) := \begin{cases} 
(CImpl, \emptyset) & \text{if } FImpl = \emptyset \\
Configure(C, FFT_n(CImpl), FImpl\{FF_n\}) & \text{if } FImpl = \{FF_1, \ldots, FF_n\} \\
\text{and } FFI_n(C) = \text{true} & \text{if } FImpl = \{FF_1, \ldots, FF_n\} \\
Configure(C, CImpl, FImpl\{FF_n\}) & \text{otherwise}
\end{cases}
\]

In analogy to Lemma 6.17, a well-formed implementation ensures that there is a bijective correlation between the valid configurations of a feature model and the possible products of a corresponding product line implementation.

**Lemma 6.17 (Equality of configured implementations)**

Let $SPLImpl = (CImpl, FImpl) = \delta(FM)$ be a well-formed product line implementation. Then it holds that two configured implementations, $Configure(CImpl, FImpl, C)$ and $Configure(CImpl, FImpl, C')$, are identical, iff the configurations $C$ and $C'$ are identical.

\[
Configure(CImpl, FImpl, C) = Configure(CImpl, FImpl, C') \Leftrightarrow C = C'
\]

**Proof:**

$\Leftarrow C = C' \text{ leads to } Configure(CImpl, FImpl, C) = Configure(CImpl, FImpl, C').$

$\Rightarrow$ The configured implementations are identical and the end results are both based on the same core frames. Therefore, the transformations that lead to the configured implementations are equal. Since a transformation function of a well-formed implementation cannot be empty (Definition 6.12) or reversible (6.14 i)) and no two features hold an identical transformation (6.14 iii)), the applied transformations must be identical. According to Definition 6.14 iv), this yields $C = C'$.

As a product line implementation is structured like a specification, the $Configure$ function can be used for obtaining the **configured implementation**. This close correlation between a product line specification and its implementation suggests that configuring and implementing a software product line are commuting operations. These commuting operations are shown in Figure 6.1. In other product line approaches often the requirements or the specification of a product line are configured (semi-)automatically and implemented manually. In our approach, the full specification of the product line is implemented using frames and allows configuration to take place on the implementational level. Obtaining the configured implementation is automated by the $Configure$ function, in case of the CoCoME product line using XVCL. This allows automatically obtaining a configured implementation for every valid configuration of the feature model.
Since the implementation using frames is straight-forward, it is possible to create a bijective function transforming the original specification into the implementation, at least theoretically. Of course, in practice the implementation usually is non-trivial and a creative refinement of a specification. Automatically transforming a specification into an implementation is only possible, if the specification already includes all information necessary for the code. Nevertheless, obtaining the product line specification from a frame-based product line implementation is possible, by merging all feature frames with an identical indicator function into one feature specification.

6.2 Product line life-cycle

The life cycle of a product line begins with its initial creation. With the ideas presented in the preceding chapters, this task consists of the following steps:

- Construct a minimal feature model. The model should be based on expected customer requirements, already planned configurations and developer experience.

- Choose a minimal configuration of the product line from the set of possible candidates. For most product lines, there is more than one candidate, as explained in Chapter 4.4. This choice can have significant impact on all further steps.

- Create a core specification from the minimal configuration and implement it. In contrast to other approaches, the resulting core of the product line is a valid configuration in this work. This allows specifying and implementing the core like a regular non-variable software system and, furthermore, allows thorough testing.

- Create feature specifications so that all variability of the feature model is covered, including added, removed and modified functionality. This also includes variable features that are members of the minimal configuration, because it must be possible to remove their functionality, if a configuration requires this.

Figure 6.1: Two commuting ways of deriving SPL members.
Implement the feature specifications using frames. For each feature specification and each modified component a frame is created, implementing the specified modifications of the respective component.

This process is depicted in Figure 6.2. Depending on the size of the initial product line, it may be reasonable to design and implement feature specifications in packages and not all at once. Usually features can be grouped easily, if they implement different choices for a feature category or if they target similar functionality. Of course, development of those groups, should not be split into several packages.

The CoCoME product line and its implementation represent a product line constructed using this process. Such a product line uses a fixed set of features from which products can be derived. This automated process is explained in Chapter 5 and is based on the configuration process described in Section 6.1. However, usually the derivation of a product is more complex. In many cases, deriving a product line member involves new functionality that was uncovered by the product line before. Also, often the functionality of a product line has to be optimized for meeting project-specific non-functional requirements.

For this reason, several product line approaches derive only the requirements documents for a configuration automatically and implement the corresponding product manually from reusable software fragments. This, however, also means that maintenance of product line assets is not necessarily propagated to already created products and depends on their manual implementation, in contrast to the fully automated approach of this work.

In the following, we focus on general steps for expanding, optimizing and maintaining a product line according to the above formalization. The general idea for extending a product line is to modify its assets without bringing the product line into an undesirable state. The
modifications applied to the product line have to be designed in such a way that the assets of the product line remain consistent. If this can be assured and the assets of an SPL are in a consistent state after the initial product line creation, evolution of the product line cannot leave its assets in an inconsistent state. This, of course, requires a formal definition of a consistent product line state, which is addressed in the following.

6.2.1 Product line consistency

Automated product derivation creates executable software products based on the valid configurations represented in a product line’s feature model. As discussed above, for a useful feature model it has to be ensured that every feature is part of at least one valid configuration. This is captured in the definition of a minimal feature model. To appropriately capture the variability of the feature model, every valid configuration of the product line has to result in a different software system. This is ensured by the definitions of well-formed specifications and implementations, as well-formedness guarantees that there is a bijective correlation between feature model configurations and the results of a configuration process (Lemmas 6.9 and 6.17).

A minimal feature model in combination with a well-formed specification and implementation allows automatically deriving a unique software product for each valid configuration. During the life-cycle of the product line, automated product derivation allows propagating changes of the product line assets to the resulting products by re-deriving their corresponding configurations. Therefore, the configurations of already derived products are of particular importance for the product line. Maintenance of the product line has to ensure that each configuration in this set of supported configurations remains valid over the product line’s life-cycle. The set of supported configurations $C_P = \{C_{P_1}, \ldots, C_{P_n}\}$ consists of at least the minimal configuration, to ensure that the core of the product line remains a valid configuration, and is extended every time a new product is derived from the software product line. If these supported configurations become invalid in the product line’s life-cycle, there is no guarantee that they can be used again to propagate modifications on the product line to the corresponding products by re-derivation.

Combining these requirements results in the definition of a consistent product line state.

**Definition 6.18 (Consistent state of a product line)**

A product line consisting of a feature model $FM = (F, fv)$, the product line specification $SPL_{Spec} = (CSSpec, FSSpec) = \delta(FM)$, the product line implementation $SPL_{Impl} = (CImpl, FImpl) = \gamma(SPL_{Spec})$ and the supported product configurations $C_P = \{C_{P_1}, \ldots, C_{P_n}\}$ are in a consistent state, iff the following properties hold:

i) $FM$ is minimal.

ii) All supported configurations are valid.

\[ \forall i \in [1..n] : C_{P_i} \in valid(FM) \]

iii) $SPL_{Spec}$ and $SPL_{Impl}$ are well-formed.

A consistent product line allows using automated product line derivation to create a unique software product for all valid configurations, including the ones that have been used for
creating products before. Starting from a consistent state, maintenance and expansion of
the product line results in a consistent state again. In the following section, we analyze the
corresponding requirements for maintaining consistency.

6.2.2 Additional functionality

When additional functionality is introduced into the product line, the assets of the product
line have to be modified to reflect this change.

**Definition 6.19 (Adding additional functionality to a product line)**

Let $FM$ be a feature model, $SPLSpec = \delta(FM)$ the corresponding product line spec-
ification, $SPLImpl = \gamma(SPLSpec)$ the implementation and $CP$ the set of supported
configurations. Additional functionality is added to the product line by mapping the
product line assets to

\[
\begin{align*}
FM & \mapsto FM' \\
SPLSpec & \mapsto SPLSpec' = \delta(FM') \\
SPLImpl & \mapsto SPLImpl' = \gamma(SPLSpec') \\
CP & \mapsto CP'
\end{align*}
\]

Additional functionality can be introduced to a product line in two ways: In form of a
variability or as a commonality. The latter simply requires altering the product line core
specification and implementation. Since the core represents a valid configuration, the new
functionality can easily be tested. As the feature model remains unchanged, the supported
configurations don’t have to be modified.

**Definition 6.20 (Adding functionality as commonality)**

Let $FM$ be a feature model, $SPLSpec = \delta(FM)$ the corresponding product line specifi-
cation, $SPLImpl = \gamma(SPLSpec)$ the implementation and $CP$ the set of supported config-
urations. Additional functionality that is not variable is added to these assets by leaving
the feature model unchanged and by modifying the specification and implementation to
change the core.

\[
\begin{align*}
FM & \mapsto FM' = FM \\
SPLSpec = (CSpec, FSpec) & \mapsto SPLSpec' = (CSpec', FSpec) \\
SPLImpl = (CImpl, FImpl) & \mapsto SPLImpl' = (CImpl', FImpl) \\
CP & \mapsto CP
\end{align*}
\]

New variable functionality is introduced by additional features. On the feature model’s
level the set of features has to be extended.

**Definition 6.21 (Additional variable functionality on the feature model level)**

Let $FM = (F, fv)$ be a feature model. Introducing additional variable functionality
requires an additional feature $F_{n+1}$.

\[
\begin{align*}
FM = (F, fv) & \mapsto FM' = (F', fv') \\
F & \mapsto F' = F \cup \{F_{n+1}\} \\
v & \mapsto fv'
\end{align*}
\]

Expanding the feature model also requires a new set of valid configurations. Supported
configurations may have to be adapted to remain valid. Depending on the number of addi-
tionally introduced constrains, the new set of valid configurations can become very complex.
However, one requirement has to be fulfilled by all product line modifications: Either the original configuration for each supported product has to be valid or the original configuration including the additional feature has to be valid. No feature may be introduced that completely invalidates any supported configuration.

**Definition 6.22 (Consistency requirement for feature model expansion)**

Let $FM' = (F',fv')$ be the extended feature model of $FM = (F,fv)$ according to Definition 6.21. The extension is conservative with respect to the additional feature $F_{n+1}$ for all previously valid configurations, iff

1. $\forall C \in \text{valid}(FM) : fv'(C) = true \lor fv'(C \cup \{F_{n+1}\}) = true$
2. $\exists C' \in \text{valid}(FM') : F_{n+1} \in C'$

Additionally, all valid configurations in $FM'$ must be also valid in $FM$ without $F_{n+1}$.

iii) $\forall C \in \text{valid}(FM') : C \setminus \{F_{n+1}\} \in \text{valid}(FM)$

This conservative extension is necessary, because otherwise supported product configurations can no longer be derived from the modified product line. Re-derivations of supported configurations are used to propagate changes made to the product line to the individual products. Condition iii) enforces that an invalid configuration for $FM$ cannot become valid by the introduction of additional functionality represented by $F_{n+1}$.

To ensure the ability to re-derive supported products, it would be sufficient to limit Definition 6.22 to condition i). The consistency definition (6.18), however, requires a feature model to be minimal, which is not guaranteed by condition i) only. This can be seen by the following lemma.

**Lemma 6.23**

Limiting Definition 6.18 to i) is not sufficient to ensure minimality in an expanded feature model. The claim $\forall C \in C_P : fv'(C) = true \lor fv'(C \cup \{F_{n+1}\}) = true \Rightarrow FM'$ minimal does not hold.

**Proof:** Consider the following feature model:

The model defines the feature set $\{A, B\}$ and the valid configurations $\emptyset, \{A\}, \{A, B\}$. Let $C_P = \emptyset, \{A\}$ be the set of supported configurations, with $\emptyset$ being the minimal configuration of the product line. Expanding the product line by introducing feature $C$ and introducing two additional constrains yields the following model:

As both configurations of $C_P$ are still valid in the extended feature model, the above equation’s left side is fulfilled. Yet, the model is not minimal, since feature $B$ is not part of any valid configuration anymore.

The extended feature model definition does not guarantee that the supported configurations $C_{P_i}$ are still valid. Because of $C_{P_i}$ being valid in the original feature model, Definition 6.22 merely ensures that either $C_{P_i}$ or $C_{P_i} \cup \{F_{n+1}\}$ is valid in the extended model. Therefore, the
supported configurations eventually have to be modified to include the additional feature for becoming valid again. If both possible configurations (the original, unmodified configuration and the extended one) are valid, the one which is closer to the original configuration is chosen. This is done to ensure that the result for re-deriving a supported configuration is as close to the original product as possible. The choice of which configuration to prefer is based on how the feature is used for modifying functionality. If the new functionality is designed and implemented as a part of the core, the new feature \( F_{n+1} \) allows removing the feature from the core. Therefore, \( C_P \) is transformed into \( C'_P := C_P \cup \{ F_{n+1} \} \). The second possibility is that the feature is designed and implemented by additional or modified feature specifications. In this case, the supported configuration remain unchanged. The choice of which design to use for adding additional functionality largely depends on expected future configurations. If it is very likely that almost any new configuration is going to include the functionality, it should be built into the core. It should be noted that since the minimal configuration is always a part of \( C_P \), it is always adapted along with all other configurations. Using the above strategies, the minimal configuration is always modified to include as little variable functionality as possible.

**Definition 6.24 (Additional functionality and supported configurations)**

Let \( C_P = \{ C_{P_1}, \ldots, C_{P_n} \} \) be the supported configurations of a product line with feature model \( FM \). Further, let \( FM' \) be the feature model of the extended product line according to Definition 6.22. In the extended product line, the supported configurations \( C'_P \) are defined by

\[
\forall i \in [1..n] : C_{P_i} \mapsto C'_{P_i} := C_{P_i} \text{ or } C'_{P_i} := C_{P_i} \cup F_{n+1}
\]

so that the extended configurations are valid in the extended product line

\[
\forall i \in [1..n] : C'_{P_i} \in valid(FM')
\]

The validity requirement in Definition 6.24 can always be fulfilled, since Definition 6.22 ensures that at least one of the two possibilities for \( C'_{P_i} \), \( C_{P_i} \) and \( C_{P_i} \cup F_{n+1} \), is valid in \( FM' \).

On the specification level, the new functionality has to be specified. For this purpose a number of modifications are required. The easiest case is to simply provide a single new feature specification that is applied, in case the new feature is part of a configuration. In many other cases, the indicator functions of present feature specifications have to be adapted, to either require the feature being part of a configuration or to explicitly require the feature not to be selected. In general, however, the required steps for including new functionality on the specification level cannot be captured.

The component-oriented nature of a product line implementation allows formulating the consistency requirements explicitly. This requires the following definition:

**Definition 6.25 (Set of applied transformations)**

Let \( SPLImpl = (CImpl, FImpl) \) be a product line specification and \( C \) a configuration. The set of applied transformations for \( SPLImpl \) and \( C \), is defined by

\[
\text{appliedT}(SPLImpl, C) := \{ FFT_i \mid (FFI_i, FFT_i) \in FImpl, FFI_i(C) = true \}.
\]

This definition captures all transformations that are applied to the core under a given specification. The set of applied features may change in the extended product line. Therefore, on the implementational level, it is more reasonable to work with applied transformations instead of applied features.
Definition 6.26 (Additional variable functionality on the implementation level)

Let $\text{SPLImpl}' = (\text{CImpl}', \text{FImpl}') = \delta(\text{FM}')$ be the extended product line implementation of $\text{SPLImpl} = (\text{CImpl}, \text{FImpl})$. Then it is required that all valid configurations $C$ result at least in the transformations that $C$ created in the unextended product line.

i) $\forall C \in \text{valid}(\text{FM}') : \text{appliedT}(\text{FImpl}, C) \subseteq \text{appliedT}(\text{FImpl}', C)$

Additionally, no transformation that removes the functionality of a pre-existing frame may be added to the implementation

ii) $\forall C \in \text{valid}(\text{FM}')$, $\forall FFT_i, FFT_j \in \text{appliedT}(\text{FImpl}', C) : FFT_i(FFT_j(\text{CImpl}')) \neq C\text{Impl}'$

Furthermore, the extended product line implementation $\text{SPLImpl}'$ can be constructed to ensure two additional properties.

I) Each of the indicator functions of the implementation evaluates to true under a valid configuration.

$\forall (\text{FFI}, FFT) \in \text{FImpl}', \exists C \in \text{valid}(\text{FM}') : \text{FFI}(C) = \text{true}$

II) The extended implementation may furthermore not introduce a duplicate transformation.

$\not\exists (\text{FFI}_1, FFT_1), (\text{FFI}_2, FFT_2) \in \text{FImpl}' : FFT_1 = FFT_2$

Condition I) can be ensured by deleting all frames with an indicator function that does not evaluate to true for a valid configuration. Obviously deleting frames that are never applied does not modify any product of the product line. In case two (or more) feature frames contain the same transformation (condition II), they can be merged. Let $(\text{FFI}_1, FFT)$ and $(\text{FFI}_2, FFT)$ be two feature frames with identical transformation functions. Then $(\text{FFI}, FFT)$ with $\text{FFI}(C) := \text{FFI}_1(C) \lor \text{FFI}_2(C)$ is semantically equivalent and eliminates a duplicate transformation from the implementation. As I) and II) are both constructible properties, only i) and ii) have to be checked manually for ensuring consistency of the extended product line assets.

Additional functionality can result in two cases, depending on whether the additional functionality was introduced as a commonality or a variability. Ensuring consistency with additional functionality in form of a commonality is easy:

Theorem 6.27 (Consistency for additional non-variable functionality)

Let $\text{FM} = (F, \text{fv})$,$\text{SPLSpec} = (\text{CSpec}, \text{FSpec}) = \delta(\text{FM})$, $\text{SPLImpl} = (\text{CImpl}, \text{FImpl}) = \gamma(\text{SPLSpec})$ and $C_P = \{C_{P1}, \ldots, C_{Pn}\}$ be the assets of a product line and let $\text{FM}' = (F', \text{fv}')$, $\text{SPLSpec}' = (\text{CSpec}', \text{FSpec}) = \delta(\text{FM}')$, $\text{SPLImpl}' = (\text{CImpl}', \text{FImpl}) = \gamma(\text{SPLSpec}')$ and $C'_P = \{C'_{P1}, \ldots, C'_{Pn}\}$
be the extended assets with additional core functionality according to Definition 6.20. If the original assets are in a consistent state, then it holds that the extended assets are in a consistent state.

Proof: For the extended assets being in a consistent state conditions i), ii) and iii) of Definition 6.18 have to hold. These conditions target the feature model, the feature specification and the feature frames. As none of these are modified in Definition 6.20, they are obviously fulfilled, if they hold for the original assets.

If additional functionality is introduced in form of a variability, ensuring the consistency of the product line assets requires a more detailed analysis. Consistency of product line assets requires a well-formed specification. Unfortunately, the modifications applied to the product line specification cannot be captured. Establishing a semantical connection between implementation and specification would allow formulating Remark 6.15 as a lemma. This would allow concluding the well-formedness of the product line specification from a well-formed implementation. Since establishing this semantical context is beyond the scope of this work, the well-formedness of the product line specification has to be postulated.

**Theorem 6.28 (Consistency for additional variable functionality)**

Let $FM = (F, fv)$,  
$SPLSpec = (CSpec, FSpec) = \delta (FM)$,  
$SPLImpl = (CImpl, FImpl) = \gamma (SPLSpec)$  
$C_P = \{ C_{P_1}, \ldots, C_{P_n} \}$  
be the assets of a product line and  
$FM' = (F', fv')$,  
$SPLSpec' = (CSpec', FSpec') = \delta (FM')$,  
$SPLImpl' = (CImpl', FImpl') = \gamma (SPLSpec')$  
$C_P' = \{ C'_{P_1}, \ldots, C'_{P_n} \}$  
the extended assets with additional functionality complying to the constraints of Definitions 6.21 to 6.24. If the original assets are in a consistent state and $SPLSpec'$ is well-formed, then it holds that the extended assets are in a consistent state.

Proof: For the extended assets being in a consistent state conditions i), ii) and iii) of Definition 6.18 have to hold.

i) If $FM$ is minimal, then all features of $FM$ are part of at least one valid configuration. This is also the case in $FM'$, including the new feature $F_{n+1}$, as this condition is part of Definition 6.22.

ii) All supported configurations $C'_{P_i}$ are valid in the extended product line according to their construction in Definition 6.24.

iii) $SPLSpec'$ is well-formed by assumption. To ensure that $SPLImpl'$ is also well-formed, conditions i) to iv) of Definition 6.14 have to hold:

i) Definition 6.26 ensures that no frame removes the functionality of another frame.

ii) The condition requires every frame indicator to evaluate to true under a valid configuration. This is ensured in Definition 6.26.

iii) Follows from Definition 6.26.

iv) Since the original product line is in a consistent state, the original implementation is well-formed. Because of Definition 6.22, all valid configurations in $FM'$ are valid in $FM$ without the new feature. According to Definition 6.26 i), such a valid configuration of $FM'$ yields at least the transformations it would yield in $FM$.
As every transformation corresponds to one feature frame (by Definition 6.14 iii)), two configurations cannot result in the same applied features if they are not equal.

Theorems 6.27 and 6.28 ensure that an expansion of the product line assets that fulfills consistency requirements 6.18 to 6.24 leaves product line assets in a consistent state, if they have been in one before the expansion. This allows automatically propagating the implemented modifications to all supported products.

6.2.3 Life-cycle tasks

Optimization A product automatically derived from a product line configuration does not always meet all requirements. Especially non-functional requirements may be very specific for one particular product. In this case project-specific optimizations need to be implemented. According to the idea of keeping the feature model and other product line assets consistent, these optimizations are a specialized case of introducing new functionality. However, since these optimizations are usually not required in any other configuration, they can be introduced more specialized than new functionality in general. Therefore, introducing an optimization is a special case of introducing additional functionality, as explained in Section 6.2.2.

The new feature \( F_{n+1} \) representing the project-specific optimization needs to be valid only under the project’s configuration \( C_P \) and this configuration has to be valid. Therefore valid\((FM'_A)\) can be defined to valid\((FM'_A) \supseteq \) valid\((FM_A) \cup \{C_P \cup \{F_{n+1}\}\})

This obviously fulfills the consistency requirements for the feature model (see Definition 6.22), as all previous configurations are still valid. Including more valid configurations (to also allow the optimization in other configurations) does not change this. On the specification and implementation levels previous assets (core and feature specifications/frames) are not modified.

The new feature \( F_{n+1} \) simply introduces one new feature specification and a corresponding set of feature frames.

It should be noted that it is possible to include more than one optimization feature. By iteratively introducing one feature per iteration the configuration to be optimized can be adapted. If two optimizations need to be introduced, the unmodified configuration \( C_P \) can be changed to \( C'_P = C_P \cup \{F_{n+1}\} \). This configuration is valid and can serve as starting point for another optimization, effectively creating valid configuration \( C_P \cup \{F_{n+1}\} \cup \{F_{n+2}\} \).

Maintenance Software product lines with automated product derivation allow automatically propagating maintenance effort to all created products. Code enhancements and bugfixes in core and feature frames are automatically forwarded to all previous products once their configurations are derived again. Usually, this process does not introduce new features and does not modify the set of valid configurations. Exceptions from this rule can arise, if (few) certain configurations require no maintenance. If maintenance, for example, introduces additional security to a component, certain configurations disallow maintenance in order to fulfill their performance constraints.

This case can also be described as ‘anti-optimization’ and is, again, a special case of introducing new functionality (see Section 6.2.2). In optimization only few configurations require modification, in this special maintenance scenario only few configurations do not have to be modified. Therefore, this scenario can be solved analogously to the optimization scenario described above. As maintenance has to be variable, feature \( F_{n+1} \) is introduced. The configurations which do not require maintenance code are expanded from \( C_P \) to \( C'_P = C_P \cup \{F_{n+1}\} \), all other configurations remain unmodified. Maintenance is then provided by a modified
core specification and exactly one additional feature specification, reversing the modifications made to the core. Therefore, including the new feature in a configuration $C$ in the extended product line creates the same products as $C \setminus \{F_{n+1}\}$ in the original product line.

The combined life-cycle of a product line can be described as a cycle of tasks, in which new functionality is introduced, optimizations are implemented and maintenance is performed. This is shown in Figure 6.3. After each iteration on the product line assets, the product line is in a consistent state and automated product derivation can be used to (re-)derive products with any valid configuration.

**6.3 Summary**

In this chapter, we formalized the concepts of our approach for designing and implementing the CoCoME product line. Based on this formal representation of a software product line, we studied constraints that have to be fulfilled when additional functionality is introduced into the product line. Taking into account these constraints, when extending, maintaining or optimizing a product line, allows keeping the product line assets in a consistent state, so that automated product derivation results in a unique software system for each valid configuration of the SPL’s feature model.
7 Conclusion and Future Work

In this work, we modeled, designed and implemented an exemplary cash desk software product line. This resulted in a new notation for designing CoBox-based variable software systems, and a general approach for implementing these designs using XVCL and automated product derivation. Based on the experience from this case study a structured process for constructing and maintaining a software product line was introduced. In contrast to other product line approaches like KobrA [ABM00] or PuLSE [B+99] this approach explicitly makes use of automated product derivation on the implementational level. To ensure that automated product derivation can be used to maintain and extend the product line over its life cycle, a set of requirements was defined the extended product line has to fulfill.

This approach bridges a gap between well-structured commercial product line approaches, which use manual product derivation, and automated product derivation. Approaches like KobrA [ABM00] and PuLSE [B+99] usually derive only requirements and the design of a product line member automatically and implement it manually by assembly of reusable software artifacts. Previous approaches using automated product derivation focus on organizational and technical requirements and provide no clear way for maintaining a product line. This gap was closed in this work, as automated product derivation is an essential tool for propagating changes in the software artifacts to the individual product line members. Expanding and maintaining the product line is defined through the requirements and the corresponding process in Chapter 6. In combination, this creates an interesting new product line approach.

7.1 Related work

There are not many findings closely related to the approach presented in this work. There is specifically no related work for the formal analysis of a product line and its assets as demonstrated in Chapter 6. However, to the general field of research it is settled in, there are some relations.

Automated Product Derivation is targeted by several publications. [McG05] discusses the organizational and technical requirements that have to be fulfilled in order to benefit from automated product derivation. Additionally, a guideline for when to expect an economic advantage from it is provided. In [CA05] an automated approach is used for deriving the design of a product line. This relates to the design notation introduced in Chapter 4.

Product line design is also discussed in [A+02]. A combination of annotated UML diagrams and decision models is used for managing design variability. This combination is extended with additional stereotype definitions in [SHJ03]. However, in both cases attention is mainly given to the general difference between common and variable aspects of the design. Resolving this variability requires additional documents. The design notation presented in this work creates a link to the feature model of the product line in the design diagrams. This provides
information about what exactly in the design is changing based on a specific product line configuration.

**XVCL-based product line implementation** Using XVCL to implement a software product line is discussed in [ZJ03], [ZJ04] and [LC04]. The case studies presented therein, however, do not provide a general approach for frame-based product line implementation. Instead, each case study uses a domain-specific frame hierarchy, which demonstrates the usefulness of XVCL’s general concepts, but is not directly applicable for other product lines.

**SPL processes** The development process described in Chapter 6 is a very basic approach. Thus, it is not easy to compare it to commercial-quality approaches like PuLSE [B+99] for example. PuLSE is a very complex methodology, customizable for a lot of different specific product line scenarios. Together with KobrA [ABM00], PuLSE represents processes for creating ‘traditional’ product lines with manual product derivation.

### 7.2 Future work

Some steps of a product line’s life cycle are not yet completely covered by the introduced approach. The requirement of conservative extensions introduced in Chapter 6, to ensure that re-deriving supported configurations is still possible, imposes rather inflexible constraints on the product line and possible extensions. As a product line usually has a longer life-cycle than its products, at some point in time it might be desirable to give up some supported configurations to achieve a higher degree of flexibility when extending the product line. Additionally, over the life-cycle of a product line its feature model cannot only be extended. Removing a feature, maybe because it is not well designed as it excludes many configurations, should be possible, as well as modifying existing features. While maintaining a product line the domain knowledge of the developing organization is likely to increase. Hence, it may be desirable to include a previously variable feature in the product line core. Making a mandatory feature variable or splitting a single feature into a set of features is possible as well. All these cases and their effects on the feature model, the specification and implementation should be investigated in the future. Additionally, the formalization presented in Chapter 6 should be extended by introducing a semantical context for product line specifications and implementations. This would allow sharpening the requirements for ensuring consistency of product line assets.

In real-world product lines the introduced consistency requirements are hard to manage manually. When extending or otherwise modifying the product line assets, a set of conditions have to be fulfilled. The complexity of these conditions increases with additional features. Real-world product lines consist of many features and optimization and maintenance, as introduced in this work, also increases this feature set. Therefore, managing and ensuring the consistency requirements presented in Chapter 6 in a more complex scenario can only be achieved with proper tool support. This also affects the up-front creation of the product line in the design and implementation phase. Although both phases can potentially be completed without tool support, it undoubtedly helps development a lot. The design notation introduced in this work uses the hiding concept that is not available in UML editors today. Keeping all occurrences of one component in the design consistent, therefore, requires a lot of manual work. A specialized editor for managing the frames of a product line implementation would
also ease development. With many frames the readability could otherwise limit an effective and fast development.

Another issue that might be overcome with proper tool support is the coordination of several development teams. In previous product line approaches, an expert team of developers builds the product line reference model and reusable assets, while one or several other teams use these assets to develop the specific products. Since these teams use the provided assets without modifying them directly, they don’t require much synchronization, if any at all. In the presented approach, however, deriving a product is done automatically, once the product line is in a consistent state. Introducing new functionality or tailoring a configuration to project-specific needs is done by iterating on the product line assets. While a single modification still leaves the product line consistent, this cannot be guaranteed if several modifications overlap. Naively introducing a new optional feature, for each of two parallel iterations, may cause problems in the feature model. If these two features are mutually exclusive, for example, this constraint cannot be introduced by any of the iterations, if they are not aware of each other. On the specification and implementation level this becomes even more complicated. Instead of working on several modifications in parallel, they have to be treated as non-interleaving transactions.

Designing and implementing these supporting tools, introducing additional semantics to the presented formalization and investigating the effects different modifications have on a product line pose interesting and challenging tasks for future work.
A Exemplary source codes

A.1 CoCoME product line constraint frame

```xml
<!-- Automatically checks constraints of the CoCoME feature model -->
<x-frame name="cocome.constraints">

<!-- Check payment constraints -->
<select option="FEATURE>PAYMENT_NONCASH">
<option value="no">
  <select option="FEATURE>PAYMENT>CASH">
    <option value="no">
      <set var="HALT_EXECUTION" value="yes"/>
      <message="Constraint not met: At least one payment option (cash or non-cash) must be checked."
        continue="yes"/>
    </option>
  </select>
</option>

<option value="yes">
  <select option="FEATURE>PAYMENT>CREDITCARD">
    <option value="no">
      <select option="FEATURE>PAYMENT>PREPAIDCARD">
        <option value="no">
          <set var="HALT_EXECUTION" value="yes"/>
          <message="Constraint not met: At least one non-cash payment option must be checked for non-cash payments."
            continue="yes"/>
        </option>
      </select>
    </option>
  </select>
</option>
</select>

<!-- Express support -->
<select option="FEATURE>SELFSERVICESUPPORT">
<option value="yes">
  <select option="FEATURE>INPUT>SCANNER">
    <option value="no">
      <set var="HALT_EXECUTION" value="yes"/>
      <message="Constraint not met: Self-service support only valid with scanner input."
        continue="yes"/>
    </option>
  </select>
</option>

<option value="no">
  <select option="FEATURE>PAYMENT_NONCASH">
    <option value="no">
      <set var="HALT_EXECUTION" value="yes"/>
      <message="Constraint not met: Self-service support only valid with non-cash payment."
        continue="yes"/>
    </option>
  </select>
</option>

<option value="yes">
  <select option="FEATURE>SINGLEDESKSYSTEM">
    <option value="yes">
      <set var="HALT_EXECUTION" value="yes"/>
      <message="Constraint not met: Self-service support only valid without single desk system."
        continue="yes"/>
    </option>
  </select>
</option>
</select>
```

82
A Exemplary source codes

Listing A.1: Constraint frame for ensuring a valid configuration declaration.

A.2 XVCL configuration file for configuring the CashDesk component

The following features are included (defined "yes") in the configuration:

- FEATURE_PAYMENT_CASH
- FEATURE_PAYMENT_NONCASH
- FEATURE_PAYMENT_CREDITCARD
- FEATURE_PAYMENT_PREPAIDCARD
- FEATURE_INPUT_SCANNER
- FEATURE_INPUT_KEYBOARD
- FEATURE_SERVERPLATFORM_WINDOWS

This results in the following configuration frame:
public void creditCardScanned(int cardNr) {
    _currentCreditCardNumber = cardNr;
    _cardReader.enablePinCode();
}

public void creditCardPinEntered(int pin) {
    _cardReader.enable();
    BankInterface bank = connectWithBank();
    if (bank == null) {
        System.out.println("CashDesk: Bank is not available, please try again or use other payment method.");
        _cardReader.enable();
        return;
    }
    // check and book
    if (!bank.validatePaymentData(_currentCreditCardNumber, pin, _currentOrder.price).await()) {
        receiveMoney(_currentOrder.price);
    } else {
        System.out.println("CashDesk: Unable to verify pin number, please try again or use other payment method.");
        _cardReader.enable();
    }
}

public void selectNonCashPayment() {
    _keyboard.setStateNonCashPayment();
    activateNonCashControllers();
}

public void activateNonCashControllers() {
    <break name="CashDesk.ActivateNonCashControllers"/>
}

public void deactivateNonCashControllers() {
    <break name="CashDesk.DeactivateNonCashControllers"/>
}

private PrepaidReader _prepaidReader;

<insert-after break="CashDesk.AdditionalMethods">
prepaidReader = new PrepaidReader(this);
</insert-after>

<insert-after break="CashDesk.AdditionalConstructor">
    _prepaidReader.enable(_currentOrder.price);
</insert-after>

<insert-after break="CashDesk.DeactivateNonCashControllers">
    _prepaidReader.disable();
</insert-after>

<insert-after break="CashDesk.AdditionalMethods">

public void prepaidCardScanned() {
    float f = _prepaidReader.getBalance().await();
    if (f >= _currentOrder.price) {
        _prepaidReader.bookValue(_currentOrder.price).await();
        receiveMoney(_currentOrder.price);
    }
}
</insert-after>
Bibliography


Bibliography


URLS are valid as of the submission date of this document (March 26, 2009).