University of Magdeburg

School of Computer Science

Bachelor's Thesis

JML-Based Verification for Feature-Oriented Programming

Author:

Jens Meinicke

April 22, 2013

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Dipl.-Inform. Thomas Thüm

Department of Technical & Business Information Systems

Meinicke, Jens:
JML-Based Verification for Feature-Oriented Programming
Bachelor's Thesis, University of Magdeburg, 2013.

Abstract

A software product line is a set of similar software products sharing a common code base. The usage of software product lines in safety-critical systems increases, but their analysis has not gained much attention so far. Generation and verification of a certain product can be done efficiently, but analysis of all products requires a large effort. This work is based on an approach to efficiently verify a whole software product line. At this approach a metaproduct represents all products of a software product line. It can be verified efficiently with the same results as to verify each product individually. However, currently this metaproduct has to be written manually, which requires high effort and is an error prone task. Based on a feature-oriented extension of Java and the Java Modeling Language we provide a tool to automatically generate a metaproduct. The metaproduct can be used to verify all products of the product line without the need to generate them. We show that our approach saved about 94% of verification time and is able to save nearly the complete time for generation. Our approach provides a solution to verify software product lines for which it is practically impossible to generate all products.

Acknowledgements

I would like to thank my advisors Thomas Thüm and Prof. Gunter Saake for giving me the opportunity to write this thesis. Without the outstanding support and advice of Thomas, this work would not be possible in this scope.

I thank Thomas Leich and the Metop for their support, which allowed me to attend the FOSD meeting at Dagstuhl that helped me to improve this thesis. I also thank all other attendees, and especially Alexander von Rhein for his support and discussion about Java Pathfinder and Model Checking.

I would like to thank Martin Hentschel for his support on MonKeY and theorem proving.

I thank Fabian Benduhn for his support on his previous work on specification and his FeatureHouse extension for contract-aware feature composition.

Finally, I thank my family and friends for all their moral support.

Contents

List of Figures x

List of Tables xi

List of Code Listings xiv

List of Acronyms xv

1 Introduction 1

2 Background 5
- 2.1 Software Product Lines . . . 5
  - 2.1.1 Feature Models . . . 6
  - 2.1.2 Feature-Oriented Programming . . . 8
  - 2.1.3 Product-Line Verification . . . 12
- 2.2 Design by Contract . . . 12
  - 2.2.1 Hoare Notation . . . 13
  - 2.2.2 Java Modeling Language . . . 14
- 2.3 Verification using Contracts . . . 16
  - 2.3.1 Runtime Assertion Checking . . . 16
  - 2.3.2 Model Checking . . . 16
  - 2.3.3 Theorem Proving . . . 18

3 JML-Based Verification of Product Lines 19
- 3.1 Design by Contract in Feature-Oriented Programming . . . 19
  - 3.1.1 Product-Line Specification . . . 19
  - 3.1.2 Contract Composition for Feature-Oriented Programming . . . 23
  - 3.1.3 Product-Based Verification . . . 27
- 3.2 Metaproduct Generation . . . 28
  - 3.2.1 Metaproduct Generation for Feature Models . . . 28
  - 3.2.2 Metaproduct Generation for Feature Modules . . . 30
  - 3.2.3 Metaproduct Generation for Specifications . . . 36
  - 3.2.4 Summary of Metaproduct Generation . . . 42
- 3.3 Family-Based Verification . . . 45
  - 3.3.1 Family-Based Theorem Proving . . . 46

3.3.2 Family-Based Model Checking 48
3.3.3 Comparison of Theorem Proving and Model Checking for Family-Based Verification 54
3.4 Summary 58

4 Tool Support in FeatureIDE **59**
4.1 Contract Composition in FeatureIDE 59
4.2 Metaproduct Generation 61
4.3 Verification of Metaproducts 63
4.4 Summary 66

5 Evaluation **69**
5.1 Case Study BankAccount 69
5.2 Time for Metaproduct Generation 70
5.3 Verification of the Metaproduct Using Theorem Proving 71
5.4 Verification of the Metaproduct Using Model Checking 73
5.5 Summary 75

6 Related Work **77**

7 Conclusion **79**

8 Future Work **81**

Bibliography **83**

List of Figures

1.1 Number of features compared to the number of possible products. . . . 3

2.1 Development costs of single systems compared to product-line engineering. 6

2.2 Feature diagram of a graph product line 8

2.3 Feature structure trees for feature modules. 10

2.4 Model checking compared to testing. 17

3.1 Domain-independent specification of software product lines. 20

3.2 Global specification of software product lines. 21

3.3 Product-based specification of software product lines. 21

3.4 Family-based specification of software product lines. 22

3.5 Feature-based specification of software product lines. 23

3.6 Feature model of the example product line. 42

3.7 Family-based model checking. 50

3.8 Merging problem with model checking of the metaproduct, with initialization of all products. 51

3.9 Family-based model-checking. 52

3.10 Merging problem with model checking of the metaproduct, with late initialization of feature selections. 53

4.1 Activation of metaproduct generation, automated verification using MonKeY, generation of all products. 62

4.2 Selection of the metaproduct generation mechanism. 64

5.1 Complete feature model for the bank-account product line. 69

5.2 Time needed for generation of the metaproduct compared to generation of all products. 71

5.3 Time need for family-based theorem proving compared to product-based theorem proving. 72

5.4 Time need for family-based model checking compared to product-based model checking, without runtime assertion checks and with parallel transactions. 74

5.5 Time need for family-based model checking compared to product-based model checking, with runtime assertion checks, and without parallel transactions. 75

List of Tables

3.1 Comparsion of verification mechanisms applied to the metaproduct. . . 57

5.1 Insertion order of features into the product line for evaluation. 70

List of Code Listings

2.1 Feature modules of the features *GPL* and *Cycle*. 10

2.2 Composition of the feature modules *GPL* and *Cycle*. 11

2.3 Constructor composition of the feature modules *GPL* and *Cycle*. 11

2.4 Example BankAccount specified with the Java Modeling Language. . . 15

3.1 Contract composition using explicit contract refinement. 26

3.2 Variability encoding for the feature model. 29

3.3 Variability encoding for the feature model with initialization of core and dead features. 30

3.4 Variability encoding for the feature model with replacement of core and dead features. 30

3.5 Metaproduct generation for method refinements. 31

3.6 Metaproduct generation for fields. 32

3.7 Initializations for fields at the metaproduct. 33

3.8 Constructor composition. 34

3.9 Metaproduct generation for branched metaconstructor composition. . . 35

3.10 Example for metaproduct generation for branched metaconstructors with compile error. 36

3.11 Metaproduct generation for flat metaconstructor composition. 36

3.12 Example for explicit contract refinement. 37

3.13 Basic contract composition for the metaproduct. 38

3.14 Check for the validity of the feature selection of the simulated product. 39

3.15 Minimum feature selections at method contracts of the metaproduct. . 39

3.16 Minimum feature selections in method contracts of the metaproduct for method overriding. 40

3.17 Contract composition for constructors at the metaproduct. 41
3.18 Handling of invariants at the metaproduct. 42
3.19 Feature module of feature *Base*. 42
3.20 Feature module of feature *Overdraft*. 43
3.21 Feature module of feature *DailyLimit*. 43
3.22 Class `FeatureModel`. 43
3.23 Metaproduct for class `Account`. 44
3.24 Initialization of the class `FeatureModel` for testing of certain products. 46
3.25 Metaproduct example of the class `Account`. 49
3.26 Feature model initialization for model checking 49
3.27 Example test case for model checking the metaproduct. 50
3.28 Feature model class for late splitting. 52
3.29 Metaproduct for model checking with contracts, avoiding initialization of features does not work it runtime assertions. 55
4.1 Generated class `FeatureModel` for Java Pathfinder. 65
4.2 Example tests for Java Pathfinder and JUnit. 66
4.3 Formal description of a proof with KeY. 66
5.1 Example test case for the feature *DailyLimit* 73

List of Acronyms

FOSD	Feature-Oriented Software Development
IDE	Integrated Development Environment
JML	Java Modeling Language

1. Introduction

Why is efficient verification of software important? In general, implementing functionality of a software product should get the main focus in a software development process, but the process of debugging and testing is highly time consuming. It is not always necessary, valuable or even possible to provide software without any bugs or only with uncritical failures. However, especially in safety-critical systems where a defect, no matter how small, can easily cause negative effects and high costs, or even harm human life, efficient verification deserves high attention. Especially when reusing software, even a small bug can cost millions [JM97]. However, reliability is one of the main goals of software engineering, especially in object-oriented programming [Mey92]. When reusing software which may be used by different kinds of applications, the reused software has to be correct [JM97].

One approach from object-oriented programming, to ensure that methods are called with right values and that the program always behaves well, is the concept of *defensive programming*. In defensive programming, assertions are introduced which check the method's parameters or other values to be correct and then decide to either execute the program or to perform some exception handling. A prominent and widely used defensive check is the check for null of the method parameters or other variables [BFL+11]. At first, such check seems to be logical in case the method cannot handle null values. However, in general this is no fault of the method, more likely it seems to be a fault of the caller. Defensive checks have not only the problem that the called method is aborted early; it is also hard to localize the real problem after introducing such checks. The more often defensive programming is used, the more often parameters must be checked. These additional and often redundant checks lead to reduced performance [APM+07, LC06], and can also lead to more errors, in case of the additional code [Mey92].

A related approach to defensive programming is the specification technique of *design by contract* [Mey92], which profits from the benefits of defensive programming, but without the disadvantages. Design by contract is inspired by one of the main ideas of

object-oriented programming, in which a program is encapsulated into smaller manageable parts, classes to define objects and methods associated with classes to define the behavior of the program. These parts can be specified with contracts, while these contracts are not part of the actual program as in defensive programming.

In design by contract, a specification describes the behavior of a method or program. A specified method can be checked, whether it behaves as defined in its specification. Contracts of methods are structured in preconditions and postcondition. A precondition defines what the method expects by a caller. Basically, the precondition describes all valid inputs of the method and the states of the program that must hold for a valid execution of the method. The postcondition defines the method's return value and the changes to the program state. Writing these specifications is associated with additional expenditure, such as for writing of comments, and can contain errors as well. However, because the contract of a method only describes the input and outcome of the method and not how the method actually works, contracts are usually simpler than the implementations.

In design by contract, the program does not abort if a condition is violated as in defensive programming, because this is usually no normal behavior. This violation should rather be used to localize the origin of this error and to fix the problem, then to simply avoid it. In contrast to defensive programming, the specifications can also be used in a static way, to check if a method is conforming to its specification.

As mentioned correctness and reliability of software is one of the main goal of software engineering, especially when reusing software [JM97]. With software product lines software can be reused efficiently. A software product line is a set of different programs that share commonalities [CN01, PBvdL05]. Product lines are already known from industry and is highly used at the automotive industry [PBvdL05]. The goal of product lines is to highly reuse existing knowledge and artifacts, to reduce costs for new products, and to use the known properties of the reused parts. At software product lines, products share same code and artifacts.

Products of software product lines share similar properties called *features* [KCH+90]. Features can be optional independent or depend on the selection of other features. Software product lines can be implemented with feature-oriented programming. Feature-oriented programming is a compositional method, where a software product is generated out of a set of feature modules [Pre97]. A feature module contains all properties or classes that belong to a feature. Feature-oriented programming is an efficient technique to generate products out of a set of chosen features.

Reliability is a main goal of software engineering. With design by contract this property can be increased. To also profit by design by contract in software-product-line engineering, previous work discussed how design by contract can be applied to feature-oriented programming [TSK+12, Ben12]. Because feature-oriented programs are implemented as feature modules, they propose compositional methods for contracts in a feature-oriented manner. They are able to generate specified programs, while the specifications correspond to the chosen features and their properties.

With design by contract applied to software-product-line engineering, the benefits of both techniques are combined. Products are highly customizable while generated products can be efficiently verified. However, the profit of high customization also comes with a major problem. With n optional, independent features a product line consists of 2^n possible products. With every new feature the number of products doubles. To verify all products of a software product line is a time consuming and up to practically impossible task.

Figure 1.1 illustrates the explosion of the number of possible products, in case of the exponential dependency to independent, optional features. As motivation we compared the number of possible products with a concrete example. For example, in a software product line with 33 optional, independent features, everyone on earth can have a unique product.

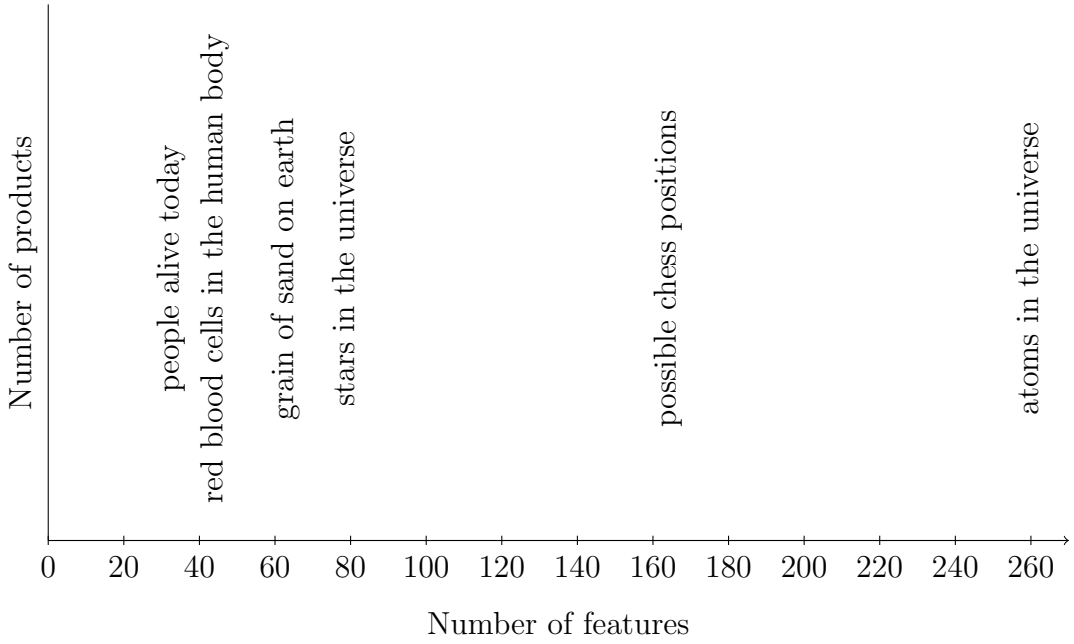


Figure 1.1: Number of features compared to the number of possible products.

The number of possible products grows up-to exponentially with the number of features. Thus, if all products of the product line need to be verified, it is not useful to verify all products in isolation. Because products share code, much code is verified redundantly. This work shows how redundant calculations can be reduces, such that all products can be verified efficiently. We generate one product that can simulate all products using runtime variability, to select the properties of the features [TSAH12, KvRE+12]. Such a product is called metaproduct.

Goal of this Thesis

The goal of this thesis is to provide tool support to automatically generate a metaproduct for feature-oriented programming using design by contract. We show how a

metaproduct can be generated for a feature-oriented software product line with design by contract. We discuss expected advantages and disadvantages for theorem proving and model checking.

We provide tool support for generation of a metaproduct. The tool is based on FeatureIDE [TKB+13], an integrated development environment for feature-oriented software development, and the composition tool FeatureHouse [AKL13], a language-independent composition tool for feature-oriented programming.

Using the automatically generated metaproduct, we were able to apply existing verification tools without any further customization. For theorem proving, we apply the tool MonKeY and as model checker we use the Java Pathfinder. Based on our approach and the tool support we were able to verify the whole software product line fully automatically, within less time compared to the separate verification of all products.

Structure of the Thesis

Chapter 2 introduces into necessary background for this thesis. In Chapter 3, we present our approach and show how a metaproduct can be generated. In addition, we discuss how different verification mechanisms can be applied, and how time for verification can be saved with our approach, compared to verification of all products. In Chapter 4, we present our tool support for metaproduct generation for feature-oriented programming using design by contract. In Chapter 5, we evaluate our approach and tool support with existing verification tools. We present related work in Chapter 6, conclude this work in Chapter 7, and discuss future work in Chapter 8.

2. Background

This chapter introduces into the essential concepts of this thesis. Section 2.1 gives a definition of software product lines. In Section 2.2 we explain the underlying specification concept design by contract. Section 2.3 gives an introduction how this concept can be used for program verification.

2.1 Software Product Lines

The need of individualized products is rising, but it comes with the price of higher costs. Mass customization is the large-scale production of goods tailored to individual customers' needs [PBvdL05]. In software engineering these individualized products are necessary, especially for databases and embedded systems with different kinds of resources and needs. A main task of software engineering is to develop strategies to reuse software.

> A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy 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 [CN01].

Software product line engineering is a paradigm to develop software applications using platforms and mass customization [PBvdL05]. A main reason of using software product lines is the reduction of development costs. Figure 2.1 shows the development cost of n single systems compared to a software product line with n systems. In literature the number of products from which it is worth to use software product line engineering, called "Break-Even Point", is around 3 or 4 systems. Product line engineering has high initial cost, but because of the high reusability the costs do not raise as fast as for developing individual systems.

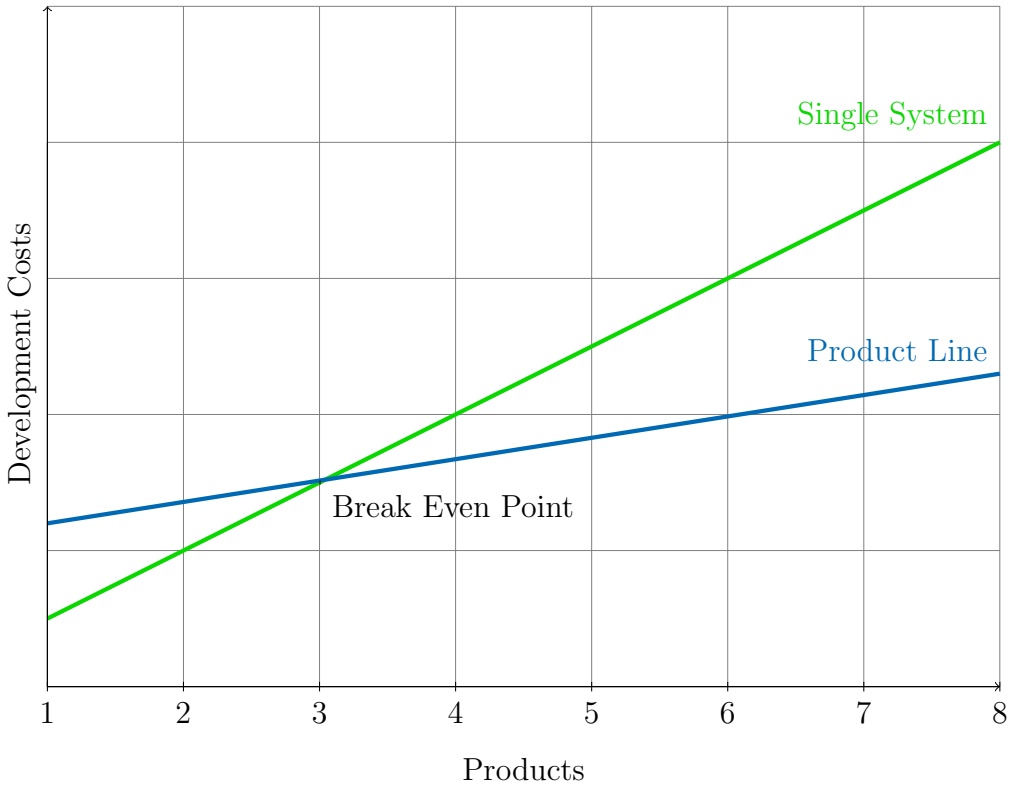


Figure 2.1: Development costs of single systems compared to product-line engineering.

2.1.1 Feature Models

Software product line engineering supports different software products sharing similar properties. These similarities and differences can be mapped into a set of features. A *feature* is a user-visible aspect or characteristic of the domain [KCH+90]. A product of a software product line is a selection of these features. Because these features cannot always be selected independently (e.g., if they represent alternative implementations), there must be constraints defining how features can be selected.

> Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised [PBvdL05].

The result of domain engineering is a feature model. A feature model represents the features of a product line and the relationships between them [KCH+90]. A feature model is an expression of propositional logic with features as variables and their relationships expressed as operators between them. These features can either be selected or unselected. A configuration of the software product line is a set of selected features while all other features are unselected. However, only the configurations, whose feature selections are a valid input for the feature model, represent valid program variants of the software product line.

Because the feature model can be complex, it is usually expressed in a *feature diagram*. A feature diagram is easier to understand and is the standard form to communicate the variability of a software product line. At a feature model there are two different kinds of features. *Concrete* features, which are mapped to implementation units and effect the program directly, and *abstract* features which are used to structure a feature model and do not have any impact at implementation level [TKES11]. Abstract features can be used to categorize a set of features into groups with similar meanings.

A feature diagram is structured as a hierarchy of features. All feature diagrams contain a root feature, and any feature can have further sub-feature. A sub-feature can only be selected if its parent feature is also selected. A sub-feature can be of different kinds. Mandatory features are always contained in any product containing also its parent feature. Optional features have an optional selection state. A sub-feature can be part of an *or-group* where at least one feature has to be selected, or part of an *alternative-group* where exactly one has to be selected. Because this hierarchy and different kinds are often not enough to describe the product lines variability there can also be additional logical formulas named *constraints* [Bat05].

The following example shows a feature model of a software product line for a graph-library Figure 2.2. The root feature *GPL* represents the base implementation of all products. All products are based on this feature. The feature model is structured into two sub-trees. The right sub-tree represents the graph type. Because any product must have a graph type this sub-tree is mandatory, marked with a filled cycle at the feature *Edge*. The feature *Edge* is an abstract feature. It is only used to structure the model, and it has no direct impact at implementation level. The sub-features of *Edge*, *Directed* and *Undirected*, define the graph type. They are in an alternative-group, marked with an empty arc at their connections to *Edge*. Because exactly one feature of an alternative-group has to be selected and the parent feature *Edge* is always selected, all products are either containing the feature *Directed* or the feature *Undirected*.

The left sub-tree of the feature model contains the features for algorithms. Because the library not necessarily need to provide algorithms, the abstract feature *Algorithm* is optional, marked with the empty cycle. The features *Number* and *Shortest*, *Cycle* are in an or-group of the feature *Algorithm*, marked with the filed arc at the connections to *Algorithm*. So, at least one of them has to be selected if the feature *Algorithm* is also selected. Because the feature *Algorithm* is optional, non of the sub-feature has to be selected, so they are just optional.

Because the algorithms for calculating the shortest path and detecting cycles need a directed graph, this must be expressed in additional logic. This logic is modeled in the constraint below the tree. If the features *Shortest* or *Cycle* are selected, then the feature Directed must also be selected. If the graph should be undirected, the features *Shortest* and *Cycle* must be unselected, too.

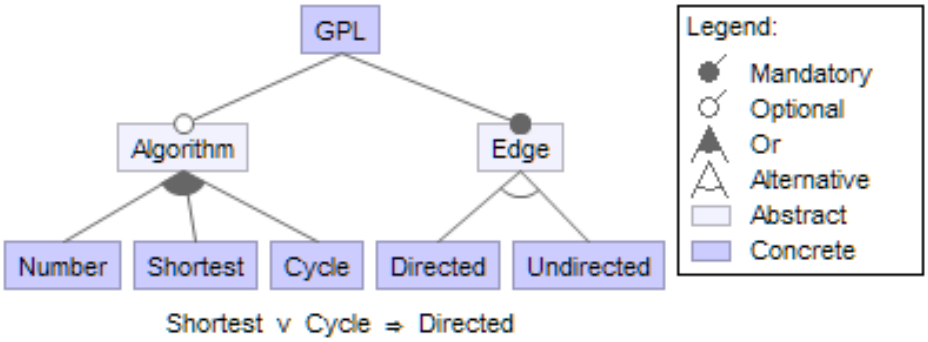


Figure 2.2: Feature diagram of a graph product line

2.1.2 Feature-Oriented Programming

With domain engineering it is possible to model relationships between features. The next step is to generate a program variant out of a set of selected features. This set of features is called configuration. To generate valid products out of a configuration, the feature selections must be a valid according to the feature model of the product line.

There are several methods to generate products of a software product line. A popular and widely used technique is to use preprocessor statements to generate products with runtime variability. The selected features are used as an input for the program while the preprocessor statements decide whether a code segment is used or not. However, the problem of using preprocessors for variability is that the program always contains all code of every feature. So the program is much larger than the useable part of the program or the actual runnable code for the given feature selection. This makes the program more complex than it has to be. Because with preprocessor annotated code it is not always clear which parts of the program are active or not, what makes the program hard to understand.

To solve this problem there are several techniques to generate a program out of a set of code segments or artifacts which are related to different feature. This process is part of *generative programming* [CE00]. The process of generation or building the program variants out of a set of program parts is called *software composition*. Mathematically, the composition of features is described with the operator $\bullet$ over the set of features F [AL08]:

$$\bullet : F \times F \rightarrow F$$

This means the composition of the artifacts of two features results a new artifact with properties of both features. A program variant p is the composition of a series of features [AL08]:

$$p = f_1 \bullet f_2 \bullet ... \bullet f_n$$

There are several techniques supporting this approach. With *delta-oriented programming* it is possible to add and remove code segments form a given code base [SBB+10]. With *aspect-oriented programming* it is possible to add properties, which cannot be encapsulated in a generalized procedure, into an existing program base [KLM+97]. However, in this thesis we focus on feature-oriented programming.

Feature-oriented programming was suggested by Prehofer as a generalization of object-oriented techniques [Pre97]. Although feature-oriented programming was invented for object-oriented languages, it can also be applied to other languages such as XML based languages or functional programming [AKL13]. Feature-oriented programming is a compositional approach where each feature can be mapped to a *feature module* which contains the software artifacts of the feature.

A way of how to compose these feature modules is superimposition. *Superimposition* is the process of composing software artifacts by merging their corresponding substructures [AKL13]. For object-oriented languages these structures are classes, interfaces and their fields and methods. In superimposition the order of features which are composed is important. First the structures of the first two features are merged to structure containing the composed elements of both features. This can also be described as that the second feature refines the first feature. If both structures contain elements for the same class, then the substructures for this class of both features are also merged. If the refining feature contains a method that is not contained in the class of the first feature, then a new method is introduced. Else, if the refining feature and the base feature contain a method with the same name and same parameter the method of the second feature could refines the other method. It is possible to replace the original method or to refine it as at the overwriting at the extension of classes in object orientation (e.g., with the `super()` call in Java). The result is a new structure of both features. This structure can be refined by a third feature, and so on.

With FeatureHouse, Apel et al. provide a language independent framework for software composition. Based on a *feature structure tree* model they are able represent any kind of software artifact in a hierarchical structure [AKL13]. With this feature structure tree it is possible to compose the feature modules to a new program with the given characteristics of the selected features.

The following example should clarify how superimposition with feature structure trees in FeatureHouse works. Figure 2.3 on the following page shows the feature structure trees for the feature modules of *GPL* and *Cycle* illustrated at Listing 2.1 on the next page. The root of a feature structure tree represents the feature for its structure. The root can have multiple sub-trees, one for each class implementation of the feature module. When sub-trees of two feature structure trees correspond to each other, they are merged. In this case, both sub-trees correspond to the same class `Graph`. So their substructures are merged to one structure for the class `Graph`.

If these classes contain sub-nodes, these nodes are merged with their corresponding composition mechanism. Fields are added or set to a given initialization value if a field with the same name already existed at the original feature module. For methods the

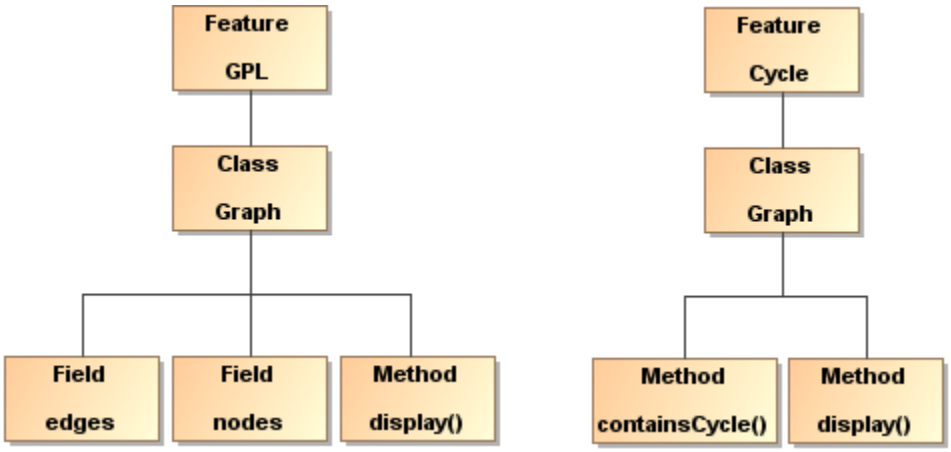


Figure 2.3: Feature structure trees for feature modules.

keyword `original()` is introduced. With this keyword it is possible to call the original method. The method `display()` of the feature *GPL* is refined by the feature Cycle. The original method needs to be renamed and is then called in place of the keyword `original()`. For this work we introduce a mechanism for renaming of methods in case of refinements. Two `$` signs are added to the methods name, to ensure that there is no method with the same name. Additionally, the name of the feature, the method that is renamed belongs to is added.

The result is a feature structured tree of the features *GPL* and *Cycle*, which can be composed with the feature structure tree of an additional feature (e.g., *Directed*). Listing 2.2 on the facing page shows the result of the composition of the features *GPL* and *Cycle*.

GPL

```
class Graph {
    Node[] nodes;
    Edge[] edges;
    void print() {
        // print the graph
    }
}
```

Cycle

```
class Graph {
    boolean containsCycle() {
        // cycle checking
    }
    void print() {
        original();
        // print the cycle
    }
}
```

Listing 2.1: Feature modules of the features *GPL* and *Cycle*.

```
class Graph {                                                    {GPL • Cycle}
    Node[] nodes, edges;
    void print() {
        print$$Base();
        // print the cycle
    }
    void print$$Base() {
        // print the graph
    }
    boolean containsCycle() {
        // cycle checking
    }
}
```

Listing 2.2: Composition of the feature modules *GPL* and *Cycle*.

In feature-oriented programming, also constructors can be refined. The mechanism for superimposition works similar to method refinements in FeatureHouse. However, in contrast to methods, constructors can only add their implementations to the existing code of this constructor. Consequently, constructors cannot overwrite existing implementations, and they do not need the keyword `original` to access the previous implementation.

The example of Listing 2.3 illustrates the composition mechanism for constructors. Both features provide implementations for the same constructor `Graph`. The composition of both constructors contains the implementations of both features. The execution order of the respective implementations is equivalent to the order the features. In contrast to method refinements, constructor refinements can directly access local variables of previous implementations.

```
class Graph {                                                          GPL
    void Graph() {
        // base initialization
    }
}
```

```
class Graph {                                                        Cycle
    void Graph() {
        // initialization of Cycle
    }
}
```

```
class Graph {                                                    {GPL • Cycle}
    void Graph() {
        // base initialization
        // initialization of Cycle
    }
}
```

Listing 2.3: Constructor composition of the feature modules *GPL* and *Cycle*.

2.1.3 Product-Line Verification

Thüm et al. [TAK+12] categorize strategies for software-product-line verification into three groups: *product-based*, *feature-based* and *family-based*. We give a brief introduction into these techniques.

Product-Based Analysis

An analysis of a software product line is called product-based, if it operates only on generated products or models thereof [TAK+12]. Product-based analysis is a quite simple technique for verification. Several products are generated and analyzed independently. All tools and mechanisms for verification can be applied easily. But the main disadvantage is that this technique can only analyze certain systems, which leads to serious problems with increasing variability and number of products.

Family-Based analysis

An analysis of a software product line is called family-based, if it (a) operates on domain artifacts and (b) incorporates the knowledge about valid feature combinations [TAK+12]. In family-based analysis the need of generating program variants is removed by analyzing all artifacts at once while considering their variability. The problem with duplicate code and redundant analysis thereof is reduced. However, the handling of family-based analysis is not as simple as for product-based. To apply existing tools and methods they need to be extended to handle the variability. In this work we show how this can be done.

Feature-Based analysis

An analysis of a software product line is called feature-based, if it (a) operates only on domain artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used) [TAK+12]. Feature-based analysis can show the validity of single domain artifact in isolation. Duplicated analysis task of product-based and family-based analysis are removed by not considering their variability. Feature-based analysis cannot show the validity of a complete product with feature interactions. However, to analyze parts of a program, this method is a good basis for a valid program.

2.2 Design by Contract

This section we give an introduction into the specification technique design by contract. We introduce into the basic concepts and benefits of design by contract. Additionally, we introduce into the Java Modeling Language, a specification language for design by contract at Java programs.

2.2.1 Hoare Notation

> How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows. [Tur49]

In 1949, Turing proposed to split a large program into smaller parts which could individually be checked with definite *assertions*. The correctness of the whole program follows by the correctness of each part [Tur49]. In general, an assertion is a true-false statement that must hold at a certain point of a program. Some programming languages, such as Java and C, support assertion checks during runtime.

Design by contract is a methodological guideline of software construction to improve the reliability or the absence of bugs [Mey92]. In design by contract, contracts between a method and its caller define the conditions the method needs for execution and the caller can expect as outcome. Formally, a method is described with two statements, a precondition and a postcondition. A Precondition is a statement about program states and method parameter that must be fulfilled when the method is called. The postcondition is a statement about program states and the return value of the method, according to the given input and program state when calling the method that must be fulfilled when the method has finished. Additionally, invariants define conditions which must be fulfilled after class initialization and also if a method is called or has finished.

A method fulfills its specification when the method is called and the precondition is fulfilled then the postcondition must also be fulfilled. When a method is called and its precondition is violated then this error is a fault of the caller. Else, if the method is called and its precondition is not violated but the postcondition is not fulfilled the error is caused by the method itself.

To describe contracts in a formal way the Hoare notation can be used. A contract c is expressed as:

$$c = \{\phi\}m\{\psi\}$$

To fulfill the contract of a method m two conditions must be fulfilled. The precondition ϕ, and the postcondition ψ if, the precondition was fulfilled, $\phi \Rightarrow \psi$.

Leavens and Cheon [LC06] described following benefits of design by contract:

Documentation. Methods can be described with an informal description, but these descriptions cannot exactly describe what can be expected from a method. Design by contract gives a formal description of all needs of a method and the expected return value.

Blame Assignment. A methods specification describes all needs of a method in its precondition, and anything a caller can expect from a method in its postcondition. If a method is called and the precondition is not hold, then the method is called in a wrong way and it cannot guarantee that the outcome is correct. In that case the caller of the method has done something wrong. If the method is called in a right way and its precondition is not violated, then the method has to guarantee that the postcondition also holds. Else, the fault is caused by the method itself. With design by contract the localization of bug is improved.

Efficiency. A possible solution to prevent methods from wrong inputs is defensive programming. In defensive programming parameters are checked whether they are a valid input, else they are not be used and the method returns immediately. However, this comes with more problems than that the method just cannot be executed. Because of these, often useless and redundant checks, the system is more concerned with checking than to execute the actual code. Also the program code is more complex and harder to understand. With design by contract this checking of parameter is no longer necessary. A method can rely on its specification and a method can also rely on the postcondition of a called method.

Modularity of Reasoning. To reason about method it is necessary to understand many other methods, but for this method the same problem appears again. It is necessary to understand also the called methods of these methods, and so on. In design by contract a method is described in a formal way. The specification is easier to understand, than the methods code, because it only describes the input and the outcome, and no implementation details. So, it is no longer necessary to understand also implementations of called methods.

2.2.2 Java Modeling Language

The Java Modeling Language (JML)[1] is a specification language that can be used to specify Java modules. With JML the concept of design by contract can be applied to Java programs. This section gives an overview about main concepts and keywords of JML.

For explanation we use the example of a bank account shown in Listing 2.4 on the next page. The account saves the actual balance. With the method `update` it is possible to deposit or withdraw money. This method returns `false` if this process cannot be done. Additionally, the balance must not be lower than the defined overdraft limit. All these informal descriptions can be defined in formal specifications using JML. JML specifications are defined in comment sections annotated with @.

Specifications in form of preconditions, postconditions or invariants start with the corresponding keyword, *`requires`*, *`ensures`* or *`invariant`* followed by the logical statement. All these statements must end with a semicolon.

[1] http://www.eecs.ucf.edu/~leavens/JML/index.shtml

At first, the balance must be higher or equal the overdraft limit. This is defined with the invariant. This statement must always hold when calling a method and also after returning from it.

In a requires clause, the minimum precondition to execute a method is defined. In the case of the update method there is no precondition, so it can always be called, so the method must handle all possible exceptions. This is defined at the requires clause that only checks for `true`. Such clauses can be removed because they are equivalent to the default JML behavior.

At the ensures clause, the methods behavior or the postcondition is defined. If the update method returns `true` then the process should have been done without any problems, else there should be no changes to the account. To access the return value of the method in a requires clause the keyword *result* can be used. For specification we also need to know the old values of fields or parameters before calling the method, this can be done with the keyword *old*.

At the first ensures clause the default behavior of the method is described. If the method returns `true`, then the balance is changed by the given parameter `x`. The second ensures clause describes the handling, if the method cannot be executed (e.g., if the balance would be lower than the overdraft limit). If the method returns `false`, then the balance has not changed. Both ensures clauses must hold, so they can also be defined in one clause.

BankAccount

```
class Account {
    //@ invariant balance >= OVERDRAFT_LIMIT;
    final int OVERDRAFT_LIMIT = 0;
    int balance = 0;
    /*@ requires true;
     @  ensures \result ==> balance == \old(balance) + x;
     @  ensures !\result ==> balance == \old(balance); @*/
    boolean update(int x) {
        /* set balance */
    }

    public /*@ pure @*/ double calculateInterest() {
        return (balance * (2 / 36500.0));
    }
}
```

Listing 2.4: Example BankAccount specified with the Java Modeling Language.

Additionally we introduce to the JML keyword *pure*. To use methods inside of clauses, it is necessary that these methods do not change the program, because this can have negative impacts on the results. To ensure that only methods without changes to the program can be used inside of contracts, such methods can be annotated with *pure*. The example of Listing 2.4 illustrats such a with *pure* annotated method. The method `calculateInterest` has no impacts on the program states, because it just calculates the interest, so it can be used inside of method contracts or invariants.

2.3 Verification using Contracts

Design by contract gives a basis of how to specify a program. Because specifications of a program should not only be used for documentation, there are several techniques to use contracts for verification. The following section describes some of them, named runtime assertion checking, model checking and theorem proving.

2.3.1 Runtime Assertion Checking

Runtime assertion checking is the most common usage of assertions. Hoare reported that about 250,000 lines of code of Microsoft Office are assertions [Hoa03] that can be checked during program execution. If the boolean expression of the assertion is false, then a violation signal is thrown [CR06]. Most compilers can enable or disable their runtime assertion checks. At design by contract the methods specifications can be used to generate assertions that can be checked during runtime.

These assertions have great benefits for debugging purpose. If a method is called with wrong parameter or it returns a wrong value, than this fault is signaled, even if this error is not visible for the user. A test can only check the outcome of one method at once, but with runtime assertions it is possible to check all called method during execution of the tested method.

Programs with runtime assertions are usually only used for debugging purpose. These programs contain a great overhead because of their additional checks. So it is also possible to compile programs without these assertions.

2.3.2 Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems [CGP99]. Clarke et al. divided the model checking process into the tasks *modeling*, *specification*, *verification* [CGP99]. The model often represents a compiled program or an abstraction thereof, which has to be verified. The specification defines properties the model must satisfy. Verification is the process to check the model against its specification.

A program consists of states and transitions. Additionally the specification is represented as logical formula. A program state describes current values of all variables, configurations, and the heap. Transactions describe possible changes from one program state to another. Model checkers examine all reachable states, while the specifications or properties are checked [DKW08]. Clarke described the model checking problem as:

> Let M be a Kripke structure (i.e., state transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M, s \models f$ [Cla08].

Model checking is an exploration of all reachable states of the program, while the states are checked against its specifications. Because model checking is a verification technique, we do not want to find all states that fulfill the specification; we rather want to find such states which do not.

Another way to describe model checking is to compare it with a simple test case. A test case only checks one path of a system at a time. According to Dijkstra: "Testing shows the presence, not the absence of bugs"[2]. To compensate this there have to be tests for every possible path, which seems to be an impossible task. However, this is exactly what a model checker can do. The program is converted into a finite state machine. A state machine can be explored while exploring all possible paths.

Figure 2.4 compares testing and model checking. On the left side there is a common test case which only executes one path. On the other side the model checker explores all possible paths until there is no path left. Testing requires a lot of experience to find all meaningful errors in a program. With model checking this problem can be solved. However, model checking comes also with a major disadvantage; the *state space explosion*. The number of possible program states and complicated data structures of current systems with many processes can be unmanageable high [CGP99].

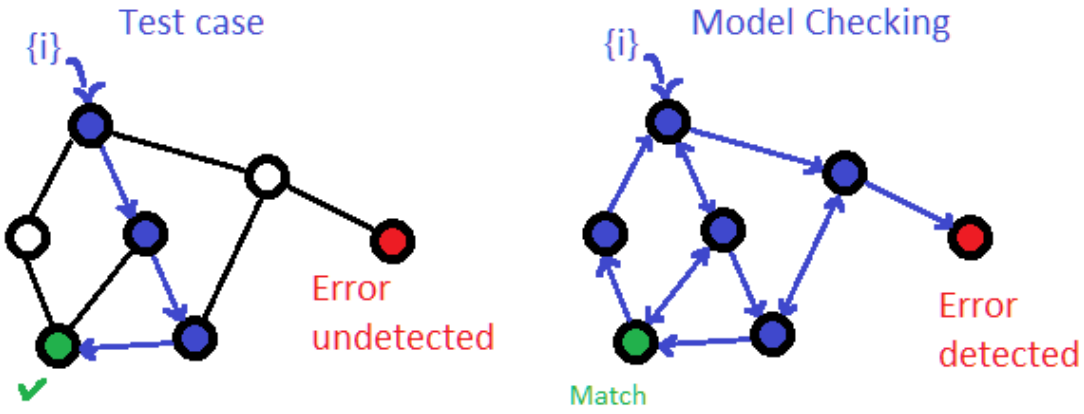


Figure 2.4: Model checking compared to testing.

During program state exploration, equivalent states can be merged, in case they are describing the same values of all variables, equivalent configurations, and have the same heap. The advantage of merging states is that all explorations from this state as root, only need to be done once. Merging of program states can reduces the number of explored states, and thus the time for verification. However, it is also necessary to save all previous explored states, what increases the required memory.

Model checking is more powerful then testing, but it generally requires more time and resources, then a simple test. To check that a method just works for a given

[2]NATO Software Engineering Conference, Rome, Italy, October 1969

input, testing is usually the better option. Because model checking is *not* equivalent to execution, the program always has to be tested.

2.3.3 Theorem Proving

Theorem proving is an deductive approach to prove the validity of logical formulas, and goes back to 1879, when Frege published his Begriffsschrift [Fre79], where he developed a formal language suitable for mathematics and logic. To verify the validity of such logical formulas, inference rules or transformation rules can be applied. A theorem prover is a tool where such inference rules can be applied to prove the validity of a formula automatically or interactively with user assistance.

Theorem proving for programs can be done using specifications in form of design by contract with JML as specification language. A prove for a method in design by contract, has the form that, if the precondition is fulfilled, then the program always executes without any errors, and then returns while the postcondition must also hold. If such a proof is proven right, then the method is conform to its specification. However, it is not simple to write specifications and especially to prove the programs with these specifications. For both, specifying and proving, highly educated, specialized experts with many experience are necessary [CGP99], especially if user interaction is necessary to complete a prove. If a statement can nevertheless not be proven, then the program is not necessarily wrong. For this reason, theorem proving comes always with other verification techniques.

Theorem Proving is time consuming and requires a lot of experience. It should only be applied if it is necessary (e.g. for safety critical systems) or beneficial (e.g. if code is highly reused in many different applications, such as libraries). However, theorem proving can only prove that a method is conform to its specification, which can also be wrong and can only describe the outcome of a method, but not other properties such as performance or memory usage. Theorem proving should not be done in isolation to verify methods. Other verification techniques, such as testing, model checking or static analysis, should always be applied in combination with theorem proving. However, if a method is proven right, the effort for other verification can be reduced rapidity.

3. JML-Based Verification of Product Lines

In this chapter, we present our approach to solve the scalability problem for verification of software product lines based on design by contract. First, we discuss available specification techniques in Section 3.1, to make a choice for our tool support. In Section 3.2, we present our approach to generate a metaproduct with JML specifications. In Section 3.3, we show how actual verification techniques can be applied to the metaproduct, to analyze the whole product line at once, and how these verification tools can profit from our approach.

3.1 Design by Contract in Feature-Oriented Programming

Design by contract is a powerful technique to specify and verify programs. There are several techniques to apply design by contract to feature-oriented programs [TAK+12, TSK+12]. In this section we give an overview how feature-oriented programs can be specified, to discuss our decision for the specification technique that matches the best for our approach to verify the whole software product line.

3.1.1 Product-Line Specification

Domain-Independent Specification

> A domain-independent specification is a specification that is usually specific to a certain programming language and is assumed to hold for all programs in that language [TAK+12].

All software products are subject of certain rules. In general, these rules are proposed by the programming language. Such rules can also be understood as specification.

Prominent examples for a domain-independent specification are syntax conformity or type systems [TAK+12]. All programs of the same programming language need to fulfill the same syntax rules.

In software product lines, all valid products need to fulfill the same domain-independent specifications. Because generation tools for software product lines have the same syntax as their corresponding compiler, they are able to find all syntax errors by parsing all feature modules separately. A prominent domain-independent check of verification tools is the check for possible null-pointer references.

Domain-independent specifications are universal for their specific types of programs. However, it is not possible to specify the actual program's behavior or to benefit from design by contract. With Figure 3.1 we illustrate a domain-independent specification. A subset of all possible programs has similar properties (e.g. they all belong to the same programming language), highlighted with red border. All these programs fulfill similar rules, such as the syntax of a given programming language or that null pointer cannot occur. Because all products of a software product line are usually of the same programming language, they have all the same domain-independent specification. The problem of domain-independent specifications is that we cannot express program specific properties as proposed by design by contract. Hence, when applying contracts to product lines, we cannot rely only on domain-independent specification.

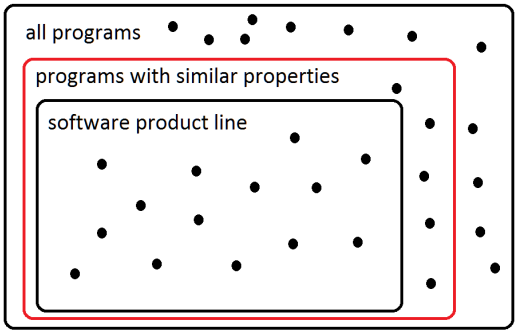


Figure 3.1: Domain-independent specification of software product lines.

Global Specification

> A global specification is a specification that all products of a software product line need to fulfill [TAK+12].

Global specifications have the advantage that specifications have only be written once for the whole product line, what reduces the effort for specification. Additionally, a programmer can always rely on the same specification, independent of the used program variant. However, only the implementation that is common to all products can be specified [TSK+12].

With Figure 3.2 we illustrate a global specification. The specification must only be written once (highlighted with red), and can be applied to all other products of the

product line (highlighted with blue). Because the specification is similar for all products, global specifications can only specify properties that hold for all products. With global specifications it is not possible to fully specify the behavior and properties of all products. Global specification is not enough for specification of product lines with contracts, this we need more sophisticated approaches [TSK+12].

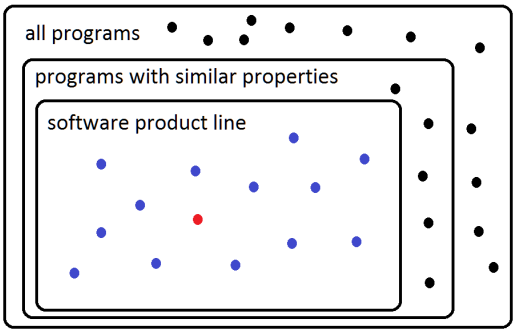


Figure 3.2: Global specification of software product lines.

Product-Based Specification

A software product line can be specified by specifying each software product individually [TAK+12]. This is called product-based specification.

As the specification for each product has to be written redundantly, this approach does not scale and should only be used for small software product lines or a productively-used subset of all products. Product-base specification is only beneficial, if the focus lies on some program variants and not all possible programs of the product line.

With Figure 3.3 we illustrate product-based specifications. Some products of the product line have their own specifications. However, all specifications are handwritten what makes it a time consuming and not useful task to specify all products, and it is usually only possible to specify a small subset of all products. With product-based specification not all products can be specified, which makes family-based verification impossible.

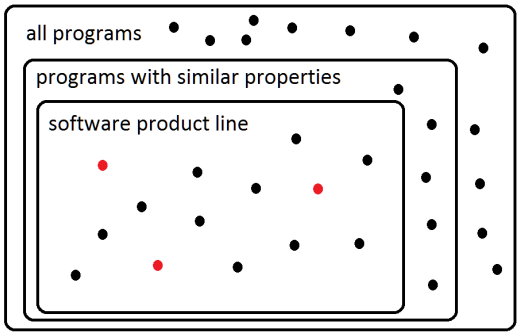


Figure 3.3: Product-based specification of software product lines.

Family-Based Specification

> A family-based specification covers the entire product line, while it has variable parts referring to individual features or feature combinations [TAK+12].

A family-based specification can be understood as an extended global specification, where some parts depend on feature selections. This technique has the benefit that it only has to be written once. With family-based specifications it is possible to generate products with specifications for the properties of this product. However, for family-based specifications it is necessary to know all features and all their interactions, to provide a specification, so that the specification with features as input specifies the corresponding product.

We illustrate in Figure 3.4 a family-based specification. The specification must only be written once, while it contains variable parts of the product line. The products of the product line are all specified with their properties. However, the specification of the product line contains knowledge of the domain, and the features cannot be specified in isolation. For specification, knowledge about feature interactions is necessary.

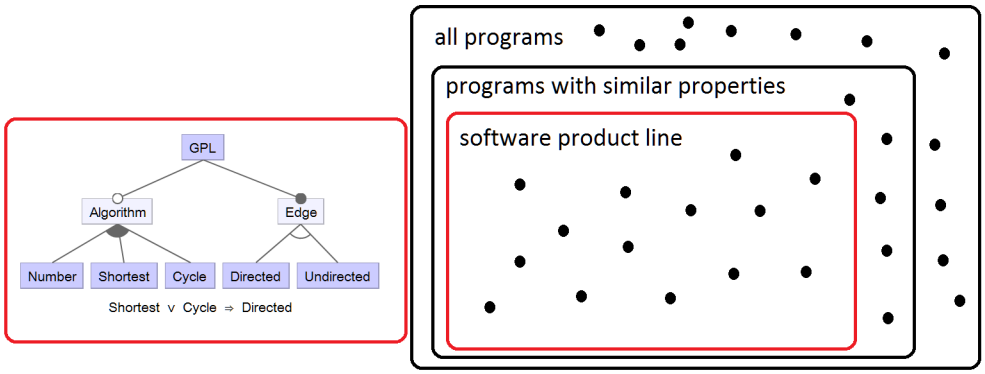


Figure 3.4: Family-based specification of software product lines.

Feature-Based Specification

Another strategy to specify software product lines is to specify each feature in isolation. *Every feature is specified without any explicit reference to other features* [TAK+12].

Specification of features or feature modules, is called *feature-based* specification. A feature-based specification is flexible and related to feature-oriented programming. In feature-oriented programming, features are implemented in separate modules. A feature-based specification is a specification of the corresponding feature modules. With specified feature modules, products can be generated with specifications, which describe the properties of the features, and thus the properties of the product.

We illustrate in Figure 3.5 a feature-based specifications. The specifications have to be written for feature modules (marked with red). When generating products out of

feature modules, also the specifications of these feature modules are composed. As result, all products of the software product line have specifications representing the corresponding features and their properties (marked with blue). In contrast to the other specification techniques, in feature-based specification there is a need for composition of the specifications, in a feature-oriented manner.

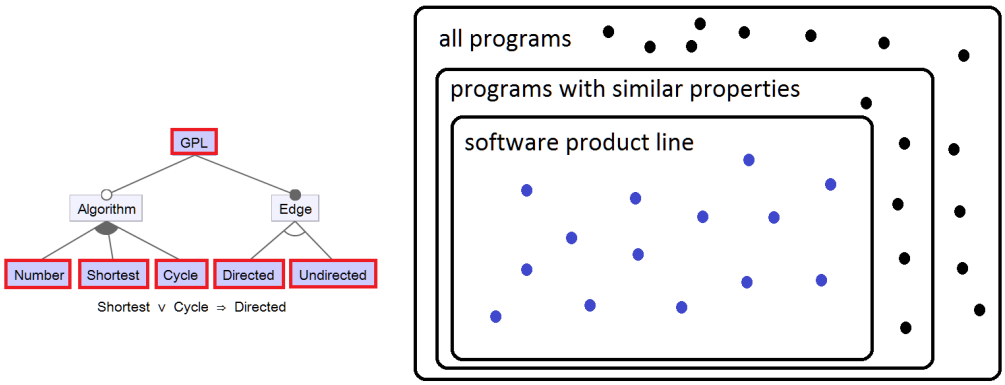


Figure 3.5: Feature-based specification of software product lines.

Compared to the previous described specification techniques, feature-based specification has several advantages:

- All products are specified with specifications which describe the properties of the product, what is not possible with domain-independent and global specifications.
- Specifications only need to be written for feature modules and not for every product as in product-based specification.
- No complete domain knowledge is necessary as in family-based specification, because specifications can be written for features in isolation.
- Specifications can be applied at code level, what makes design by contract possible, in contrast to domain-independent specification.

3.1.2 Contract Composition for Feature-Oriented Programming

Feature-based specification comes with several advantages. To apply design by contract to software product lines using feature-oriented programming, feature-based specification can be used efficiently. Because methods can be refined in feature-oriented programming, also their contracts must have composition techniques. In this section, we discuss several composition techniques for method contracts of feature-oriented product lines.

Plain Contracting

> In plain contracting, any method introduction may be specified with a contract. *Method refinements can only change the implementation of a method, but not its contract.* Thus, method refinements are not specified with contracts. The developer must ensure that any method inside a given feature module is either a method refinement in all products or a method introduction in all products [Ben12].

Plain contracting is the simplest of all composition techniques, because only method introductions can be specified [TSK+12]. With this approach, it can be ensured that a implementation of a method can be changed by refinements, but a caller can always rely on the same specification. With plain contracting a programmer can use methods without considering possible refinements. However, plain contracting is simple and intuitive, it is not possible to apply this composition technique to all product lines [TSK+12], because method refinements often require contract refinements.

Consecutive Contract Refinement

> When using consecutive contract refinement, method introductions and method refinements may be specified with a contract. The intended meaning of a method refinement including a contract is that *the new contract must be fulfilled in addition to the original contract.* For the resulting method both contracts must be fulfilled [Ben12].

Consecutive contract refinement is related to behavioral subtyping [DL96] in object-oriented programming, in which a subtype need to fulfill its specification and also the specification of the supertype. In this approach, a programmer can always rely on the introduced contract, also if the method is refined. However, it is not always possible to express both contacts in one contract [Ben12].

Contract Overriding

> When using contract overriding, method introductions and method refinements may be specified with a contract. *Method refinements override the contract of the original method.* Thus, each method refinement must include a contract that completely specifies the behavior of the method after refinement [Ben12].

In contract overriding, method refinements only need to fulfill the specification they are introducing, but with this simple technique to refine there come some disadvantages. First of all, a caller cannot rely on a specification because it can be overwritten. Moreover, because contracts cannot refer to the specification of the refined method, it is often necessary to copy parts of the overwritten specification, which leads to clones in specifications.

Pure Method Refinement

Contracts can include method calls, in JML marked as `pure`. In an approach for feature-oriented programming named *pure method refinement* [TAK+12, Ben12], these pure methods can be refined such that this refinement also changes the specification. These pure methods can use the underlying programming language to describe a contract, but with the disadvantage that some commands of the specification language are no longer available, such as `old`.

Explicit Contract Refinement

> When using explicit contract refinement, method introductions and method refinements may be specified with a contract. Method refinements override the contract of the original method. Thus, each method refinement must include a contract that completely specifies the behavior of the method after refinement. [Ben12]

To aware that the methods implementation matches its specification when refining methods, also the method contracts can be refined. In explicit contract refinement, the problems of the previously discussed contract composition mechanisms are solved. Original contracts can be referenced at the refining contract so the previous contracts do not have to be cloned. However, it is still possible to override the original contract. In contrast to plain contracting, it is possible to change specifications in a way that the composed contract describes the behavior of both features. In explicit contract refinement it is also possible to use pure methods and refinements thereof, to express additional information. All previously discussed contract composition mechanisms can be encoded using explicit contract refinement. Explicit contract refinement has the benefits of all other contract composition mechanisms, but can also be adjusted with additional restrictions to express other mechanisms. Because explicit contract refinement is a flexible and powerful mechanism applying design by contract to feature-oriented programming, we rely on this mechanism in this thesis.

For explicit contract refinement, we need a mechanism for contract composition. Therefor the keyword `original` [TSK+12] is introduced as an extension for JML. If a method is refined in a way that also the contract of the refined method should hold then the keyword `original` can be used instead of cloning the original contract, such as needed for contract overriding. With the keyword `original` it is possible to reference the corresponding clause of the refined contract. That means if `original` is used in the requires clause, then the requires clause of the original contract must also hold. The same holds for the ensures clauses. If `original` is not used in a clause, then the clause of the original contract is overwritten with the new clause.

The following example (Listing 3.1 on the following page) illustrates how contract composition using explicit contract refinement with the keyword `original` works. The example represents a bank account with a field saving the current balance and a method

to update this value. The update method of the feature *Base* is refined by the feature *Withdraw*. The specification of the feature *Base* ensures that after a successful update, the current balance of the account has changed by `x`. Else, if the update cannot be performed for some reason, than the balance must not be changed.

Base

```
class Account {
    int balance = 0;
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    boolean update(int x) {
        /* set balance */
    }
}
```

DailyLimit

```
class Account {
    int withdraw = 0;
    /*@ ensures \original
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw)); @*/
    boolean update(int x) {
        /* set withdraw */
        original(x);
    }
}
```

{*Base* • *DailyLimit*}

```
class Account {
    int balance = 0;
    int withdraw = 0;
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x)
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw));@*/
    boolean update(int x) {
        /* set withdraw */
        update_Base(x);
    }
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    private boolean update_Base(int x) {
        /* set balance */
    }
}
```

Listing 3.1: Contract composition using explicit contract refinement.

At the feature *Withdraw* the amount that is removed from the account is saved using the field `withdraw`. The specification of the update method ensures that only the amounts that are removed from the account are saved at the field `withdraw`. If an update can not be performed, `withdraw` does not change. We see that the specification of the method update only describes the feature's implementation without considering the original specification. Because the method `update` of the feature *withdraw* is refining

the base implementation, also the contract of the original method must hold in this case. To express this, the keyword `original` is used at the ensures clause.

The third part of the example shows the result of composition of the features *Base* and *Withdraw*. The keyword `original` is replaced by the referenced requires clause of the feature *Base*. The composed contract of the update method specifies the composed implementation of update.

3.1.3 Product-Based Verification

Explicit contract refinement is a flexible and powerful way to specify a feature-oriented software product line. With this technique it is possible to generate products with specifications representing their feature selection. An analysis that operates on such a product is called *product-based* [TAK+12]. Product-based analysis has the main benefit that existing tools can be easily applied to the generated product. Standard tools and analysis such as those provided by compilers can be applied without any additional customization. Product-based verification is a simple strategy to analyze products and is clearly a good method for small software product lines or software product lines where only a subset of all possible products need to be verified [TAK+12].

However, if all products of the product line need to be analyzed, product-based analysis comes with major disadvantages. If we assume a product line with 32 independent features, there are 2^{32} valid products, and if we assume generation and verification for a product least together just 1 second, then it would take 136 years to complete this task. This process can be easily parallelized and the time for analysis can be reduced by more powerful computers.

After analysis we may identify several errors, but the process of debugging an handling of all program variants and errors leads to new problems, especially is human interaction is necessary. In feature-oriented programming, the actual implementation work should be done at the feature modules. Hence, a propagation of all errors from variants to the feature modules is necessary [TKB+13], which also lasts time, because it also depends on the exponential number of products. After fixing an error, it is necessary to ensure that this error does not appear in other program variants, and to ensure that with this fix no new error is introduced, the whole process needs to be done all over again.

To compensate the problem of scalability there are methods which only operate on subsets of all products. These methods are also called *optimized*, in contrast to generating all products, which is called *unoptimized* [TAK+12]. Because analyzing a subset of all products cannot always have the same results as analyzing all products, there is a need to efficiently select products to cover as many errors as possible with this limited number of products. One strategy is to focus on feature interactions. Errors can be detected which are caused by interactions between two features. This technique is also called *pair-wise* verification [OMR10]. As generalization of this approach there are strategies which focus on interactions between t features, also called *t-wise* [PSK+10].

Product-based verification, whether optimized or unoptimized, has two main problems. If it is necessary to verify all possible products, product-based strategies do not scale

in case of the exponential number of products. Additionally, because products of a software product line share the same code, many analyses are done redundantly. This holds especially for core implementations that all products share.

Product-based strategies are simple and a beneficial way to verify software product lines with few products and the preferred way to analyze product lines where only some products need to be analyzed. But if all possible products should be analyzed, product-based strategies have their limits, especially if the software product line can be configured by a user. This work focuses on solving the problems of scalability and redundant analysis.

3.2 Metaproduct Generation

The idea to solve the problems with redundant calculations and exponential number of products is to analyze all artifacts once while considering variability defined in the feature model or directly at code level. This approach is called *variability-aware* verification or *family-based* verification [TAK+12]. One family-based strategy is to generate a metaproduct that contains all code fragments of all features. The variability of the product line is encoded as runtime variability [TSAH12, KvRE+12]. The variable parts of the program are executed while considering the feature selections of a configuration as input. The generated product is also called *product simulator*, because it can simulate all possible products with the corresponding configuration as input. The idea of this strategy is that existing verification tools can handle this metaproduct and its variability without a need for adapting those tools.

Based on previous work of Thüm et al. [TSAH12], this work focuses on this family-based strategy. We show how a metaproduct can be generated for a given product line out of feature modules to be able to simulate all possible products of the product line using runtime variability. Thereby, we also consider contracts. Because explicit contract refinement seems to be the most expressive contract composition mechanism, we use this mechanism for contract refinements. We show how runtime variability can be introduced into contracts, so that the simulated contracts specify the simulated products.

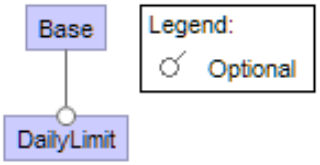
3.2.1 Metaproduct Generation for Feature Models

The metaproduct must exactly simulate all possible and valid products of the software product line. To ensure that only valid feature selections according to the given feature model are considered during analysis, we need a *variability encoding* of the feature model. Thus, we need a class containing any information saved at the feature model. It is necessary to use a normal class, so that the verification tools can handle it as part of the program. To be able to simulate all valid products, the class representing the feature model needs two things:

1. One field for every feature saving its selection state. The selection of these features represents one concrete configuration, which can then be used to simulate an actual product.

2. A method that calculates the validity of the current selection. This method contains the logic of the feature model to decide whether the current feature selection is possible. This method ensures that only possible products are analyzed.

We use the following example, to explain the details of the encoding of a feature model. Listing 3.2 represents the class for the illustrated feature model. Both features *BankAccount* and *DailyLimit* have a boolean variable. They are set to `true` if they are selected, and `false`, otherwise. The method `valid()` calculates the validity of the current feature selection. To calculate the validity, we use the CNF representation of the feature model expressed with standard Java symbols. The method needs to be annotated as *pure*, so it can be referenced in constraints. All fields and the method need to be `static`, so they can be used in other classes without handling an actual object of the class `FeatureModel`. The class can be used to set the configuration as input and simulate the corresponding program variant, but this is not the intention of this class. The idea is to let the fields be undefined, so that verification tools can analyze all valid feature combinations. Section 3.3 described how this actually works for theorem proving and model checking.



Metaproduct

```
public class FeatureModel {

    public static boolean dailylimit, base;

    public /*@ pure @*/ static boolean valid() {
        return base && (!dailylimit || base);
    }
}
```

Listing 3.2: Variability encoding for the feature model.

For optimization of the verification process, the class `FeatureModel` can also contain additional information calculated before generation. A feature model can contain dead features, which can never be selected, so their value can initially be set to `false`. A feature model can also contain core features which are contained in all configurations. These core features can initially be set `true`. At the example of Listing 3.3 the feature *Base* is a core feature, so it can be initialized with `true`. To ensure that such precalculated feature selections never change, the fields for dead and core features need to be `final`.

Because initialized fields for core and dead features are not variable, it is also possible to replace all their occurrences with the concrete value. In Listing 3.4, we illustrate

how all occurrences of the field `base` are replaced with `true`. This handling can also have further benefits, as we discuss later.

FeatureModel

```
public class FeatureModel {

    public static boolean dailylimit;
    public final static boolean base = true;

    public /*@ pure @*/ static boolean valid() {
        return base && (!dailylimit || base);
    }
}
```

Listing 3.3: Variability encoding for the feature model with initialization of core and dead features.

FeatureModel

```
public class FeatureModel {

    public static boolean dailylimit;

    public /*@ pure @*/ static boolean valid() {
        return true && (!dailylimit || true);
    }
}
```

Listing 3.4: Variability encoding for the feature model with replacement of core and dead features.

With the representation of the feature model as part of the program, verification tools are able to use them during analysis. We described how to simulate a configuration. In the next parts we show how runtime

3.2.2 Metaproduct Generation for Feature Modules

With the described class `FeatureModel` we have an encoding for the feature model. This class can be used to set the feature selections to simulate a configuration. In this section, we show how this information can be used to introduce runtime variability into the metaproduct, such that the metaproduct behaves exactly as the simulated product.

The mechanisms for methods with runtime variability is already described in detail for method refinements in previous work [ASW+11, TSAH12]. This work shows how runtime variability can also be introduced into other parts of the code, such as constructors and fields.

Metaproduct Generation for Methods

Apel et al. [ASW+11] proposed a transformation for feature modules, especially for methods, into a metaproduct. Thüm et al. [TSAH12] described how to generate a metaproduct out of feature modules for feature-oriented programs using the feature selections from the described class `FeatureModel`. With dynamic branching at method

refinements, the behavior according to the feature selections can be simulated. When calling a refining method, the corresponding feature selection is checked. If the feature is selected, then the composed code is used. Otherwise, if the feature of the refinement is not used, then the previous implementation is used instead.

Listing 3.5 illustrates this mechanism on the `BankAccount` example based on the previous discussed class `FeatureModel`. The method `update` of the feature *DailyLimit* refines the corresponding method `update` of the implementation of feature *Base*. However, in contrast to standard composition where the composed method of both features would be executed, the selection of the feature *DailyLimit* is checked first. If the value of the field `dailylimit` is `true`, then the composed method is executed. Otherwise, if it is `false`, then only the previous implementation is used, in this case the implementation of the feature *Base*. This mechanism is used every time a method is refined. With this mechanism all possible feature combinations can be simulated.

Base

```
class Account {

    boolean update(int x) {
        /* set balance */
    }
}
```

DailyLimit

```
class Account {
    boolean update(int x) {
        /* set withdraw */
        original(x);
    }
}
```

Metaproduct

```
class Account {
    boolean update(int x) {
        if (FeatureModel.dailylimit) {
            update$$Base(x);
            return;
        }
        /* set withdraw */
        update$$Base(x);
    }

    private boolean update$$Base(int x) {
        /* set balance */
    }
}
```

Listing 3.5: Metaproduct generation for method refinements.

Metaproduct Generation for Fields

Methods can refine existing implementations, but fields are composed differently. When the same field is provided by multiple features with different initializations, then only the initialization of the last feature is used [AKL13]. However, at the metaproduct, we need the initializations of all features. To implement variable initialization of fields according to the feature selection, we use the ternary operator `?` (Java syntax) for the conditional expression, which is equivalent to an if-then-else statement, to select the corresponding initialization statement.

The example (Listing 3.6) illustrates the difference between the standard composition mechanism for fields and the handling of fields at the metaproduct. In the base implementation, the fields for the maximum overdraft is set to `0`. The optional feature *Overdraft* sets the limit to `-5000`. If both features are composed then only the initialization of *Overdraft* is used. In contrast, the metaproduct must simulate both initializations depending on the selection of the feature *Overdraft*. The initialization depends on the selection of the feature. If the feature is selected then the value of the if statement is `true` and the refinement is used, and otherwise, the field is set to the value of the previous feature.

Base
```
class Account {
    int OVERDRAFT_LIMIT = 0;
}
```

Overdraft
```
class Account {
    int OVERDRAFT_LIMIT = −5000;
}
```

{*Base* • *Overdraft*}
```
class Account {
    int OVERDRAFT_LIMIT = −5000;
}
```

Metaproduct
```
class Account {
    int OVERDRAFT_LIMIT  = FeatureModel.overdraft ? −5000 : 0;
}
```

Listing 3.6: Metaproduct generation for fields.

The described mechanism for field composition at the metaproduct, works in general. However, we discovered scenarios in which the metaproduct would not behave as the simulated product. For the description of the discovered problems and our solutions, we use the example of Listing 3.7. At the implementation of feature *Base*, the field `OVERDRAFT_LIMIT` is not initialized directly. However, fields of a primitive data type, such as `int` or `boolean`, are initialized with default values. At the metaproduct, the default initialization must be used if a field of a primitive data type is not initialized explicitly. In this case, if the feature *Overdraft* is not selected, then the field `OVERDRAFT_LIMIT` is initialized with `0`. A similar behavior must be applied for

objects, but in contrast to primitive data types they must be initialized with `null`. At the example, the field `id` is not initialized at *Base*, and is set to `1` at the feature *Overdraft*. To simulate both initialization, the field must be set to `null` if the feature *Overdraft* is not selected.

A field introduction, such as `transactions` at the feature *Overdraft*, can call constructors or methods for initialization, what changes the simulated program, if the field does not exist at this program. In Listing 3.7, the feature *Overdraft* introduces the field `transactions`, which does not exist at programs without the feature *Overdraft*. To simulate that fields are only initialized if the corresponding feature is also selected, the field is set to `null` if the feature is not selected. In our example, we removed the initialization with `null` for the feature `Base`, because this feature is mandatory in all products.

Base
```
class Account {
    int OVERDRAFT_LIMIT;
    Integer id;
}
```

Overdraft
```
class Account {
    int OVERDRAFT_LIMIT = -5000;
    Integer id = 1;
    List transactions = new List();
}
```

Metaproduct
```
class Account {
    int OVERDRAFT_LIMIT  = FeatureModel.overdraft ? -5000 : 0;
    Integer id = FeatureModel.overdraft 1 ? null;
    List transactions = FeatureModel.overdraft ? new List() : null;
}
```

Listing 3.7: Initializations for fields at the metaproduct.

Metaproduct Generation for Constructors

For method refinements the proposed approach with dynamic branching can be applied, because all method refinements introduce separate methods. However, constructors are refined with another mechanism. After constructor composition there is only one constructor containing all refinements (see Section 2.1.2). Listing 3.8 illustrates how constructors are refined. In constructor composition refinements cannot override. They always extend the previous implementations. So there is no need for the keyword `original`, as known from method refinements, to call the original constructor. In the example, the implementation for `Account()` of the feature *DailyLimit* is executed directly after the previous implementation of the feature *Base*.

To simulate all possible constructor of all products, we discuss two approaches (Listings 3.9 on page 35 and 3.11 on page 36). A difference to method composition is that

Base

```
class Account {
    int balance;
    Account() {
        balance = 0;
    }
}
```

DailyLimit

```
class Account {
    int DAILY_LIMIT;
    Account() {
        DAILY_LIMIT = 1000;
    }
}
```

{*Base* • *DailyLimit*}

```
class Account {
    int balance;
    int DAILY_LIMIT;
    Account() {
        balance = 0;
        DAILY_LIMIT = 1000;
    }
}
```

Listing 3.8: Constructor composition.

constructor refinements can access local variables of implementations they are refining (e.g., the constructor refinement of *DailyLimit* can use local variables of the base constructor).

The problem of local access to implementations of previous constructors, is solved by the composition approach presented at Listing 3.9. The constructor has one branch for each feature selection, hence we call it *Branched Meatconstructor Composition*. If the feature *DailyLimit* uses a local variable of the feature *Base*, than it can have access in this case. However, there is also a branch where *Base* is not selected. In this particular example, the implementation of feature *DailyLimit* cannot have access to the variable of feature *Base*, because the variable does not exist, which causes a compile error. Although this path can never be executed because the selection for this branch is invalid, and generally such code can only be executed if there is also a product with the same problem. However, we assume compile error free product lines, because compile errors can be detected efficiently by type checkers [AKGL10, KA08]. To solve the compile error, in case of invalid local access, these paths can be removed by calculating all valid paths with the given feature model. Another problem of this branched metaconstructor composition is that the generated constructor of the metaproduct grows exponentially with every new refinement. However, it is unlikely that every feature refines the same constructor.

Metaproduct

```
class Account {
    int balance;
    int DAILY_LIMIT;
    Account() {
        if (FeatureModel.base) {
            balance = 0;
            if (FeatureModel.dailylimit) {
                DAILY_LIMIT = 1000;
            }
        } else {
            if (FeatureModel.dailylimit) {
                DAILY_LIMIT = 1000;
            }
        }
    }
}
```

Listing 3.9: Metaproduct generation for branched metaconstructor composition.

The example in Listing 3.10 illustrates the problem of compile errors when accessing local variables. We introduced the local variable `local` at the constructor of feature *Base* and the constructor refinement of feature *DailyLimit* uses this local variable. At the branch, where *Base* is unselected and *DailyLimit* is selected, the local variable is undefined, what causes a compile error. However, such errors are only possible for branches, which do not represent a valid configuration or the same compile error is also part of the products for these configurations. To solve this compile error problem, the branches that represent invalid configurations can be removed, so this error can only happen if there are products with the same error.

Branched metaconstructor composition comes with several problems. The size of the constructor grows exponentially, and to solve compile error problems additional calculations are necessary. To solve the discussed problems, we provide another technique named *Flat Metaconstructor Composition*, where access to local variables between constructor refinements is forbidden. Listing 3.11 illustrates how the second mechanism works. The generated constructor contains all implementations from the feature modules, but only executes each implementation if the corresponding feature is selected. The refinement of the feature *DailyLimit* cannot access local variables of the base implementation, but all other problems of the previous approach cannot appear. The generated constructor can still simulate all possible feature combinations.

Flat metaconstructor composition, comes with several advantages, in contrast to generating one branch for all possible feature combinations. If there is no local access to variables, then the expressiveness of both mechanisms is equivalent, but the size of the flat metaconstructor raises only linear. Additionally, the constructor is far simpler, what makes verification and analysis easier and avoids redundant verification. However, if local access is necessary the composition mechanism with branches must be used.

Base
```
class Account {
    Account() {
        int local = 0;
    }
}
```

DailyLimit
```
class Account {
    Account() {
        local++;
    }
}
```

Metaproduct
```
class Account {
    Account() {
        if (FeatureModel.base) {
            int local = 0;
            if (FeatureModel.dailylimit) {
                local++;
            }
        } else {
            if (FeatureModel.dailylimit) {
                local++;/* local variable is undefined */
            }
        }
    }
}
```

Listing 3.10: Example for metaproduct generation for branched metaconstructors with compile error.

Metaproduct
```
class Account {
    int balance;
    int DAILY_LIMIT;
    Account() {
        if (FeatureModel.base) {
            balance = 0;
        }
        if (FeatureModel.dailylimit) {
            DAILY_LIMIT = 1000;
        }
    }
}
```

Listing 3.11: Metaproduct generation for flat metaconstructor composition.

3.2.3 Metaproduct Generation for Specifications

With the described techniques it is possible to generate a product that can simulate all possible products by setting the feature variables according to the configuration of the

product. Because we want to profit from design by contract for verification, also the method contracts and class invariants need to be variable, so they always specify the simulated product. In this section, we show how runtime variability can be introduced into contracts and invariants, to simulate specifications for a given feature selection.

Metaproduct Generation for Method Contracts

In explicit contract refinement, contracts can be refined, to adjust the specification. The metaproduct can simulate all programs, using runtime variability with a feature selection as input. Hence, also contracts need to specify the simulated program using the same feature selections as input.

For illustration we use the known example for explicit contract refinement at Listing 3.12. Based on the work of Thüm et al. [TSAH12] and the described mechanism for method composition for metaproducts, a method's metaspecification consists of two alternative specifications. One specification must hold if the refining feature is selected, which is generated by composing the specifications of the previous feature and the refining feature. If the refining feature is not selected, then the specification of the previous feature must hold. In the example, the contract of the composition of *Base* and *DaylyLimit* must hold if the product for both features is simulated. However, if the feature *DailyLimit* is not selected and the product for the feature *Base* is simulated, then only the contract of the feature *Base* must hold.

Base

```
class Account {
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    boolean update(int x) { ... }
}
```

DailyLimit

```
class Account {
    /*@ ensures \original
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw)); @*/
    boolean update(int x) { ... }
}
```

{*Base* • *DailyLimit*}

```
class Account {
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x)
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw));@*/
    boolean update(int x) { ... }
    }
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    private boolean update_Base(int x) { ... }
}
```

Listing 3.12: Example for explicit contract refinement.

Listing 3.13 illustrates how the contract of the metaproduct can simulate the corresponding contract of the simulated product. The clauses of the method `update` depend on the feature selection and which product should be simulated. If the product where the feature *DailyLimit* is unselected is simulated, then the contract for previous feature, in this case *Base* must hold. This activation with the condition that the feature *DailyLimit* is unselected is expressed with the implication.

```
!FeatureModel.dailylimit ==> clause(base)
```

Using an implication has the benefit that the clause is always true if the feature *DailyLimit* is selected, so the clause for unselection of the feature must only be checked if the feature is not selected. For selection of a feature the activation of the clause works similar.

```
FeatureModel.dailylimit ==> clause(base • dailylimit)
```

The clause for the composition of *Base* and *DailyLimit* is only checked if the feature *DailyLimit* is selected. If *DailyLimit* is not selected, then only the clause of feature *Base* is checked.

Metaproduct

```
class Account {
    /*@ ensures !FeatureModel.dailylimit ==>
     @   (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x);
     @  ensures FeatureModel.dailylimit ==>
     @   ((!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x)
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw)));@*/
    boolean update(int x) { ... }
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    private boolean update$$Base(int x) { ... }
}
```

Listing 3.13: Basic contract composition for the metaproduct.

For method contracts we removed the check for the selection of the feature *Base*, which is in this case not necessary, because all configurations contain this feature. However, in general, all features of contract refinements must be checked, to know which clauses are active and which are not.

Additional to the contracts of the feature modules, it is necessary to check some properties that must hold when a method is called. When analyzing the metaproduct it must be guaranteed that only valid configurations are simulated. To only simulate valid products, the method `valid` of the class `FeatureModel` is added as requires

```
class Account {                                                    Metaproduct
    /*@ requires FeatureMdel.valid();
     @  ensures !FeatureModel.dailylimit ==> ... ;
     @  ensures FeatureModel.dailylimit ==> ... ;@*/
    boolean update(int x) { ... }
    /*@ requires FeatureMdel.valid();
     @  ensures ... ; @*/
    private boolean update$$Base(int x) { ... }
}
```

Listing 3.14: Check for the validity of the feature selection of the simulated product.

clause to all method contracts. Listing 3.14 illustrates the additional requires clause for the validity of the simulated product.

With the described rules, a metaproduct can simulate products and their corresponding contracts. Additionally, only valid products are considered at the method contracts. However, if methods are called, it is also necessary to know the products in which this method exists. This information of feature selections is especially necessary to know how fields are initialized, how other variable methods must behave, and what invariants are active, which we discuss later. Listing 3.15 illustrates how this feature selection is expressed at requires clauses of method contracts. To call the method `update`, at least the feature *DailyLimit* or the feature *Base* has to be selected. Additionally when the method `update$$Base` is executed the feature *Base* has to be selected.

```
class Account {                                                    Metaproduct
    /*@ requires FeatureMdel.valid();
     @  requires FeatureModel.dailylimit || FeatureModel.base;
     @  ensures !FeatureModel.dailylimit ==> ... ;
     @  ensures FeatureModel.dailylimit ==> ... ; @*/
    boolean update(int x) { ... }
    /*@ requires FeatureMdel.valid();
     @  requires FeatureModel.base;
     @  ensures ... ; @*/
    private boolean update$$Base(int x) { ... }
}
```

Listing 3.15: Minimum feature selections at method contracts of the metaproduct.

The described mechanism only works, if all method refinements contain the keyword `original` in their body. If a method refinement does not call the original implementation, then the original implementation does not appear at products where this refining feature is selected. To specify that renamed methods are never called in such cases, we add a precondition which requires that this feature is unselected to all previous method contracts. Listing 3.16 illustrates the differences for the contract of the base implementation, if the update method of the feature *DailyLimit* would not call the original implementation of the feature *Base*. The contract requires that feature

DailyLimit is not selected, because this implementation of the feature *Base* does not exist in configurations where the feature *DailyLimit* is selected.

Metaproduct

```
class Account {
    /*@ requires FeatureMdel.valid();
     @  requires FeatureModel.dailylimit || FeatureModel.base;
     @  ensures !FeatureModel.dailylimit ==> ... ;
     @  ensures FeatureModel.dailylimit ==> ... ;@*/
    boolean update(int x) { ... }
    /*@ requires FeatureMdel.valid();
     @  requires FeatureModel.base;
     @  requires !FeatureModel.dailylimit;
     @  ensures ... ; @*/
    private boolean update$$Base(int x) { ... }
}
```

Listing 3.16: Minimum feature selections in method contracts of the metaproduct for method overriding.

Metaproduct Generation for Contracts of Constructors

Contracts for constructors are composed similar to method contracts at the metaproduct. The clauses are active if the corresponding features are selected. Additionally, contracts for constructors at the metaproduct need the call of the method `valid` to ensure that only valid products are simulated. As for methods, we need to know for which feature selections the constructor exists, so we add a minimum selection of features whit a requires clause.

Listing 3.17 on the facing page shows the result of a contract composition for constructors in the metaproduct. At the base implementation, the initial balance is set to `0` and the feature *DailyLimit* initializes the field DAILY_LIMIT with `1000`. The contract can simulate all valid feature combinations. If the feature *DailyLimit* is selected then the composed contract must hold, else, only the contract of the feature *Base* has to hold.

Base

```
class Account {
    int balance;
    /*@ ensures balance == 0; @*/
    Account() {
        balance = 0;
    }
}
```

DailyLimit

```
class Account {
    int DAILY_LIMIT;
    /*@ ensures \original && DAILY_LIMIT == 1000; @*/
    Account() {
        DAILY_LIMIT = 1000;
    }
}
```

```
class Account {                                          {Base • DailyLimit}
    int balance;
    int DAILY_LIMIT;
    /*@  ensures balance == 0 && DAILY_LIMIT == 1000; @*/
    Account() {
        balance = 0;
        DAILY_LIMIT = 1000;
    }
}
```

```
class Account {                                                  Metaproduct
    int balance;
    int DAILY_LIMIT;
    /*@ requires FeatureModel.valid();
     @  requires FeatureModel.dailylimit || FeatureModel.base;
     @  ensures !FeatureModel.dailylimit ==> balance == 0;
     @  ensures FeatureModel.dailylimit ==>
     @   (balance == 0 && DAILY_LIMIT == 1000); @*/
    Account() {
        if (FeatureModel.base) {
            balance = 0;
        }
        if (FeatureModel.dailylimit) {
            DAILY_LIMIT = 1000;
        }
    }
}
```

Listing 3.17: Contract composition for constructors at the metaproduct.

Metaproduct Generation for Invariants

Feature modules can introduce additional invariants (see Section 2.2.2). To ensure that invariants must hold, if they actually exist, invariants must be conditional. An invariant is active when the corresponding feature is selected. Again, we express this by means of an implication.

The handling of invariants is illustrated by the example in Listing 3.18. In the feature *DailyLimit*, the withdrawn money must not be higher than `1000`. This condition is expressed with the additional invariant. Because this condition must only hold in product with this feature, the invariant is only active in the metaproduct if *DailyLimit* is selected.

```
class Account {                                                  DailyLimit
    /*@ invariant withdraw <= DAILY_LIMIT;@*/
    int DAILY_LIMIT = 1000;
    int withdraw = 0;
}
```

```
class Account {                                                    Metaproduct
    /*@ invariant FeatureModel.dailylimit ==> withdraw <= DAILY_LIMIT;@*/
    int DAILY_LIMIT = 1000;
    int withdraw = 0;
}
```

Listing 3.18: Handling of invariants at the metaproduct.

3.2.4 Summary of Metaproduct Generation

For simplification we discussed all parts of metaproduct generation in isolation. For illustration how the generation concepts for method refinements, constructor refinements, field refinements and introductions, constructor refinements and invariant introductions, work together, we give an example metaproduct at this point.

The example product line consists of three features a total of 4 different program variants. The corresponding feature model is illustrated at Figure 3.6. Each feature has its own feature module illustrated at Listings 3.19, 3.20 and 3.21. All feature modules refine the class `Account`.

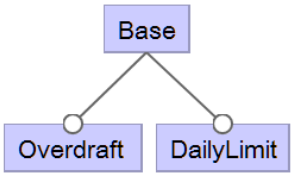


Figure 3.6: Feature model of the example product line.

```
public class Account {                                                  Base
    final int OVERDRAFT_LIMIT = 0;
    //@ invariant balance >= OVERDRAFT_LIMIT;
    int balance;
    /*@ ensures balance == 0; @*/
    Account() {
        balance = 0
    }
    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x); @*/
    boolean update(int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = newBalance;
        return true;
    }
}
```

Listing 3.19: Feature module of feature *Base*.

Overdraft
```
class Account {
    final int OVERDRAFT_LIMIT = −5000;
}
```

Listing 3.20: Feature module of feature *Overdraft*.

DailyLimit
```
class Account {
    //@ invariant withdraw >= DAILY_LIMIT;
    final static int DAILY_LIMIT;
    int withdraw = 0;
    /*@ ensures \original && DAILY_LIMIT == −1000; @*/
    Account() {
        DAILY_LIMIT = −1000;
    }
    /*@ ensures \original
     @   && (!\result ==> withdraw == \old(withdraw))
     @   && (\result ==> withdraw <= \old(withdraw)); @*/
    boolean update(int x) {
        int newWithdraw = withdraw;
        if (x < 0)  {
            newWithdraw += x;
            if (newWithdraw < DAILY_LIMIT)
                return false;
        }
        if (!original(x)) return false;
        withdraw = newWithdraw;
        return true;
    }
}
```

Listing 3.21: Feature module of feature *DailyLimit*.

Metaproduct
```
public class FeatureModel {
    public static boolean dailylimit, base, overdraft;

    public /*@pure@*/ static boolean valid() {
        return base && (!overdraft || base) && (!dailylimit || base);
    }
}
```

Listing 3.22: Class `FeatureModel`.

At first, the class representing the feature model needs to be generated. Listing 3.22 illustrates the corresponding class `FeatureModel` for the feature model of the product line. All features have a field defining their selection state. Additionally, the method `valid` represents the logic for valid feature combinations.

The metaproduct that is able to simulate all 4 program variants is illustrated at Listing 3.23. The field `OVERDRAFT_LIMIT` is introduced at feature *Base* and refined at feature *Overdraft*. If the feature *Overdraft* is selected, then the refined value is used, else,

the original value of *Base* is used. The selection of *Overdraft* has also impacts on the invariant of feature base, because the field `OVERDRAFT_LIMIT` is used in this invariant.

The feature *DailyLimit* introduces a invariant. To ensure that this invariant is only checked for products with the feature *DailyLimit*, the invariant is only active if the field for the feature *DailyLimit* is `true`. The feature *DailyLimit* also introduces the field `withdraw`. This filed must not be initialized with a default value, because the filed is not refined. However, a conditional initialization is necessary for field introductions, if the data type is not primitive.

At the constructor `Account` and the method `update` all combinations of refinements can be simulated. Additionally, the composed contracts are active in case of the selection of the feature *Base* the feature *DailyLimit*. If the feature *Base* is selected and the feature *DailyLimit* is not selected, the only the contracts of the original implementations are active. Additionally, it is necessary that the methods and constructors exist and that the combination of feature selections is valid, what is expressed with the additional requires clauses. With a detaild look at the implementation of the method `update$$Base` we see that also the selection of the feature *Overdraft* has an impact on the method `update`, because the field `OVERDRAFT_LIMIT` which is conditional is used.

Metaproduct

```
public class Account {
    /*@ invariant (balance >= OVERDRAFT_LIMIT); @*/
    final int OVERDRAFT_LIMIT = FeatureModel.overdraft ? -5000 : 0;
    int balance;
    /*@ invariant FeatureModel.dailylimit ==>
     @      (withdraw >= DAILY_LIMIT); @*/
    final int DAILY_LIMIT;
    int withdraw = 0;
    /*@ requires FeatureModel.valid();
     @  requires FeatureModel.dailylimit || FeatureModel.base;
     @  ensures !FeatureModel.dailylimit ==> balance == 0;
     @  ensures FeatureModel.dailylimit ==>
     @      balance == 0 && DAILY_LIMIT == -1000; @*/
    Account() {
        if (FeatureModel.base) {
            balance = 0;
        }
        if (FeatureModel.dailylimit) {
            DAILY_LIMIT = -1000;
        }
    }
    /*@ requires FeatureModel.valid();
     @  requires FeatureModel.dailylimit || FeatureModel.base;
     @  ensures !FeatureModel.dailylimit ==>
     @  !\result ==> balance == \old(balance)
     @  && \result ==> balance == \old(balance) + x;
     @ ensures FeatureModel.dailylimit ==>
     @  !\result ==> balance == \old(balance)
     @  && \result ==> balance == \old(balance) + x
     @  && !\result ==> withdraw == \old(withdraw)
     @  && \result ==> withdraw <= \old(withdraw); @*/
```

```
    boolean update(int x) {
        if (!FeatureModel.dailylimit) {
            return update$$Base(x);
        }
        int newWithdraw = withdraw;
        if (x < 0)  {
            newWithdraw += x;
            if (newWithdraw < DAILY_LIMIT) return false;
        }
        if (!update$$Base(x)) return false;
        withdraw = newWithdraw;
        return true;
    }
    /*@ requires FeatureModel.valid();
     @  requires FeatureModel.base;
     @  ensures (!\result ==> balance == \old(balance))
     @  && (\result ==> balance == \old(balance) + x); @*/
    private boolean update$$Base(int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT) return false;
        balance = newBalance;
        return true;
    }
}
```

Listing 3.23: Metaproduct for class `Account`.

3.3 Family-Based Verification

The metaproduct can be used to simulate all valid products of a software product line, while the contracts specify the simulated product. A metaproduct is designed for verification rather than as a replacement of products. The purpose of the metaproduct is to analyze all valid products at once, without generating and analyzing all products, which is a time consuming, redundant, and often even impossible task. In this section, we discuss how a metaproduct with contracts can be verified with model checking and theorem proving. We also show how the time for verification can be reduced using the metaproduct, compared to verification of all products, while both strategies have the same outcome.

We show how the metaproduct can save time for testing of certain products. A test case can only cover and test one configuration at once. To use the metaproduct to simulate and test a certain product, the corresponding configuration must be set before executing the test case. This can be done by initializing the fields of the class `FeatureModel` with concrete values. Listing 3.24 on the next page shows how a concrete product can be simulated and tested using the metaproduct. When executing the test, one configuration is simulated at once. After initialization, the method `valid` can be used to verify that a valid configuration is used. Instead of covering only one configuration also iteration over all valid configurations or a subset thereof is possible. With this

mechanism, concrete products can be tested without generating and compiling them. Compared to testing only one product, testing the metaproduct is not beneficial, but if the metaproduct is used to test multiple products, testing the metaproduct can save time. When testing the metaproduct, a test suite is valuable that covers several products and several test cases. Additionally, testing using the metaproduct can also profit by runtime assertions, especially blame assignment is beneficial.

```
public class AccountTest {
    public void test() {
        FeatureModel.base = true;
        FeatureModel.dailylimit = false;
        assertTrue(FeatureModel.valid());
        /* test case */
    }
}
```

Listing 3.24: Initialization of the class `FeatureModel` for testing of certain products.

3.3.1 Family-Based Theorem Proving

With theorem proving, it is possible to prove that a method fulfills its specification. Such a proof shows that, if a method is called and its precondition is not violated, than the method must execute without throwing unexpected errors. Additionally, the method must hold its postcondition after it has finished. If a method's contract is proven to be fulfilled, then it is ensured that the method fulfills its specification.

At the metaproduct, a theorem prover assumes that the values for feature selections, defined at the class `FeatureModel`, are initially unknown (except if values of core and dead features are set). All features can be selected or unselected. Because a theorem prover assumes that all fields that can be changed, have unknown values, it is necessary to prove the method for all possible states of the fields. Applied to the fields for the features defined at the class `FeatureModel`, a theorem prover checks a method for all possible combinations of the values of these fields. To only analyze valid feature combinations, we add the method `valid` to each contract as requires clause. We also add a minimum selection of features as requires clauses, to know whether the method exists at this product (see Section 3.2.3). With these information in form of additional requires clauses, the theorem prover only analyze valid products, and thereby knows which clauses of contracts are active, which invariants are active, and how fields are initialized.

Using Implementations Instead of Contracts

In theorem proving, contracts can be used in different ways. The first way is to verify a method and its contract, while all contracts of called methods are ignored and their implementation is used instead. This technique is called *method inlining* [BHS07]. The benefit of *method inlining* is that contracts of called methods do not need to be specified,

because they are simply ignored. However, the problem is that the proof can be very large, because any called code by this method is part of the proof.

There are several reasons why a theorem prover is not able to proof a method:

- The method does *not* fulfill its contract.
- The method does fulfill its contract, but violates other restrictions, such as possible null-pointer references.
- The method does fulfill its specification and is correct, but the theorem prover is not able to prove this.

At a certain product we know the configuration causing this problem. Because the metaproduct simulates all products at once, blame assignment is not that easy because we do not know which configurations cause the problem. However, it is necessary to locate the features causing the problem. If a method is only introduced in one feature and is not refined by further features, we know that the feature introducing the method is causing the problem or is at least part of it. However, methods can be refined and introduced by several features. We know the feature causing the problem for unrefined methods. If this method is refined by a further feature, then there are two proves, because there are two methods with their corresponding contract at the metaproduct. If the method that can simulate all combinations of feature selections can be proven, then it is unnecessary to also prove the refined method. The refined method is already part of the proof for the method that refines. However, if the simulating method cannot be proven, we need to prove also the previous method. If the method introduction of the first feature is proven to be correct, then the problem is caused by the refinement, else, the error is caused by the method that is refined.

For clarification of blame assignment at the metaproduct we use the example of Listing 3.13. First, the method `update` is proven. If this method `update` is proven to be correct, then also the method `update$$Base` must be correct, because this method and also its contract can be simulated by the method `update`. If the method `update` cannot be proven, then we need to know whether the method `update$$Base` can be proven. If the base implementation can also not be proven, then we know that the feature *Base* is part of the problem. If `update$$Base` can be proven, then the refining feature *DailyLimit* causes the problem.

Using Contracts Instead of Implementations

Another way using contracts in theorem proving is to replace all method calls by their contracts [BHS07]. At this work we call this technique *using contracts*. When verifying a method, then the implementation of this method is used, but all method calls are replaced by their contracts, except if they are not specified, then their implementation is used.

There are several benefits of *using contracts*. The prove does not raise as at *method inlining*. Called methods of called methods do not need to be considered, because they are part of the used contract. With this smaller proves also the time for verification can decrease. However, a major disadvantage of *using contracts* is that all used contract must describe the behavior of the method and its changes to the system exactly enough that the theorem prover can use them instead of the implementation itself. *Using contracts* has great benefits, but it also harder to use then *method inlining*.

With *using contracts* for theorem proving, all methods are verified in isolation. To ensure that a method is proven to be correct, it is not enough that the proof for this method is closed. In contrast to *method inlining*, it is also necessary that all called methods are also proven to be correct. However, if a method cannot be proven then the blame assignment works exactly the same as for *method inlining*.

Using contracts has not only potential in software development, but also for verification of the metaproduct. After changing code of a program it is only necessary to prove the methods and contracts that have changed, all methods that call a method whose contract has changed, and contracts that use changed pure methods, because all other methods are already proven. The initial proof is a hard and time consuming task, but if it done once, all further proofs must only be done for single methods. The same holds for the metaproduct. If the metaproduct is proven once, then the proof after a change can be faster than completely prove one certain product.

3.3.2 Family-Based Model Checking

In software model checking, a program is considered as a graph of program states, and transitions as possible changes from one program state to another (see Section 2.3). The graph can split into different paths at certain points. These points can be variables with random values (e.g., a filed storing a boolean value). Applied to the metaproduct, the values of the fields for each feature need to be variable, and the graph should be explored while considering the variability defined by the class `FeatureModel`.

Model Checking of the Metaproduct without Contracts

If a field has different values then there is one state at the graph for each value. To simulate that the fields for feature selections at the class `FeatureModel` can have different values, we use the method `random()`, which returns either `true` or `false`. The model checker generates states and transitions for both values. One state represents the selection of the feature, while at the other state the feature is unselected. To apply this method to the metaproduct, all fields for features at the class `FeatureModel` can be initialized directly. After initialization, there is one state for each possible combination of feature selections. However, because only valid products need to be verified, only states representing valid feature combinations, must be checked. Therefor the method `valid` of the class `FeatureModel` is called directly after initialization of all fields. If the method `valid` returns `false`, then the state does not represent a valid product, so we can skip transition starting in this state. Hence, if the state

represents a valid product, then the model checker can analyze the product represented by this state. Because all fields for features at the class `FeatureModel` are initialized with concrete values, the metaproduct simulates exactly the product represented by this state.

For illustration we use the known example of the bank account at Listing 3.25. If the method `update` is called, then the model checker should analyze both possible products. With the class `FeatureModel` at Listing 3.26, all combinations of the features *Base* and *DailyLimit* are possible. Because the fields are initialized with random values, there is one path at the graph for each combination for the selections of all features.

A model checker needs a starting point at the program. This can be the main method or a test case. Such a test case is illustrated at Listing 3.27. To ensure that only valid feature combinations are checked, the validity of the feature selection is evaluated at the beginning of the test. If a valid feature selection is given, the actual test can be executed. In this case the method `update` is called. At this point the graph has one path for each valid feature combination. Hence we can check all products without generating each product.

Metaproduct

```
class Account {
    boolean update(int x) {
        if (FeatureModel.dailylimit) {
            update$$Base(x);
            return;
        }
        /* set withdraw */
        update$$Base(x);
    }

    private boolean update$$Base(int x) {
        /* set balance */
    }
}
```

Listing 3.25: Metaproduct example of the class `Account`.

Metaproduct

```
public class FeatureModel {
    public static boolean base = random();
    public static boolean dailylimit = random();

    public /*@pure@*/ static boolean valid() {
        return base  &&  (!dailylimit  ||  base);
    }

    boolean random() {
        return Math.random() <= 0.5;
    }
}
```

Listing 3.26: Feature model initialization for model checking

```
public class AccountTest {                                    Metaproduct
    @Test
    public void updateTest() {
            if (!FeatureModel.valid()) return;
            Account a = new Account();
            a.update(100);
            assertEquals(100, a.balance);
    }
```

Listing 3.27: Example test case for model checking the metaproduct.

Figure 3.7 illustrates the graph for initializing all products at the beginning of the test. All possible feature combinations are generated because of the reference to the class `FeatureModel`. After initialization of all features, the invalid combinations are ignored, because the method `valid` is called at the beginning of the test case. The paths which are not ignored can be handled as normal products, because all features have concrete and fix values.

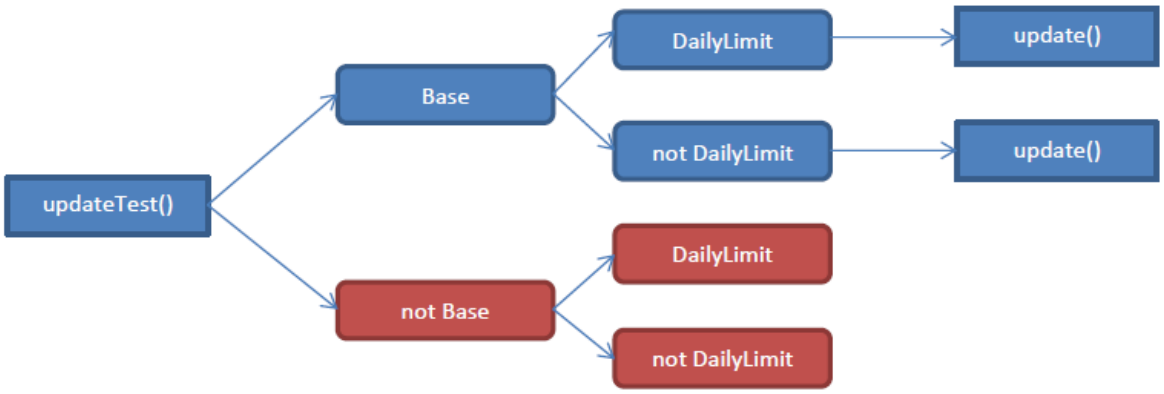


Figure 3.7: Family-based model checking.

The first problem of this approach is that there is on path for each feature combination, which are up-to 2^n paths for n features. This causes the graph to grow exponentially. Even if only valid configurations are considered, which minders the problem slightly, there is still one path for each valid product. The second problem is that methods or implementations that are equivalent in different products are checked multiple times redundantly [KvRE+12].

Figure 3.8 illustrates the problem of merging between states representing different products, in case that the fields of the class `FeatureModel` for feature selections are set initially. After initialization of all features, there is one independent graph for each possible feature combination. Inside of such a graph representing one product, merging is possible. However, merging between different products is not possible, because the fields for feature selections at the class `FeatureModel` have different values. The problem is that a major part of analysis is done redundantly, because the graphs representing different products are equivalent in many cases.

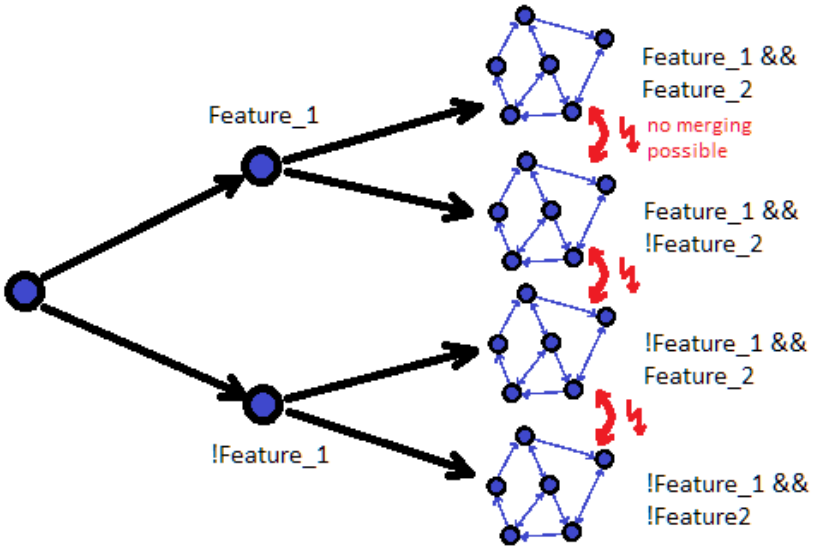


Figure 3.8: Merging problem with model checking of the metaproduct, with initialization of all products.

A better solution is to only split paths at the points where the products differ. The paths should be split at the points where the feature selection is checked and different paths are executed. At this point, there is one path for the selection of the checked feature and one path where the feature is unselected. In this approach, the features must not be initialized at the beginning. Initially, they need to have unknown values. When their selection state is checked the first time, the graph needs to be split into two paths for this feature selection. Because only valid configurations must be considered and some values can still be unknown, it is necessary to check whether the current selection state can lead to a valid configuration, else this path must not be checked.

For illustration of the described approach we use the bank account example again. Listing 3.28 on the next page represents the corresponding class `FeatureModel`. To simulate the unknown selection of the features, we use the class `Boolean`, while `null` represents the unknown selection state. Instead of calling the fields directly, a corresponding method is called instead. Each feature has its own method, in this case `dailylimit()` and `base()`. These methods first request, whether the corresponding feature has a known selection state. If the value is not `null`, then the current selection is returned, else, if the value is still unknown, the graph splits into two paths at this point, one path for each selection state. After splitting it is necessary to only analyze valid configurations. Therefore the partial configuration of each path is checked, whether it can lead to a valid configuration. If the partial configuration is invalid, the corresponding path is ignored. If a valid configuration is possible, the path is analyzed and the feature selection is set and returned.

Figure 3.9 on the following page illustrates the graph, based on the example of Listing 3.27, where the path is only split at the positions where runtime variability changes the

programs behavior. In this case, the graph is split at the point where the selection of the feature *DailyLimit* is checked at the method `update`. The field of *DailyLimit* is set for both program variants, so there are two states for these two products. After initialization of the field for *DailyLimit*, the corresponding code for the simulated product is checked.

```
public class FeatureModel {
    public static Boolean base, dailylimit;

    public /*@pure@*/ static boolean valid() {
        /* leads to valid configuration */
    }

    public static boolean dailylimit() {
        if (dailylimit == null) {
            dailylimit = random();
            valid();
        }
        return dailylimit;
    }
    public static boolean base() {
        ...
    }
}
```

Listing 3.28: Feature model class for late splitting.

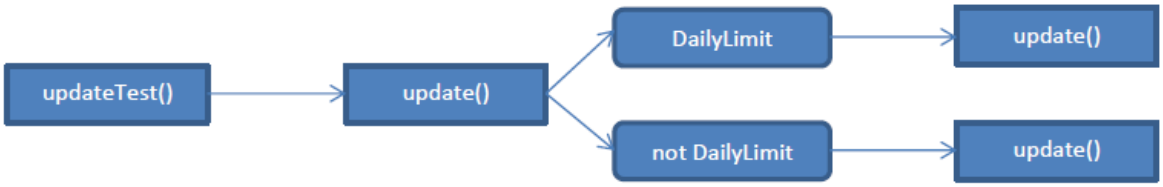


Figure 3.9: Family-based model-checking.

Figure 3.10 illustrates how the graph of program states and transitions can be merged, if features are not initialized until their selection is checked. Initially the graph is equivalent to a graph of a certain program. Just if a features selection is checked, in case of runtime variability, there are two graphs for the section of this feature, at the example the selection of *Feature_1*. Between these two graphs it is not possible to merge, because the field for *Feature_1* at the class `FeatureModel` differs. However, as long as there is no additional selection state of other features is checked, merging inside of these graphs is possible. If a selection of an unset field of a feature is checked, then there are two new graphs for this new partial configuration.

Model checking of the metaproduct with late splitting, highly depends on the number of used features. If only few features must be set, then the time for model checking is reduced. Another way to reduce the time for model checking is to always set known

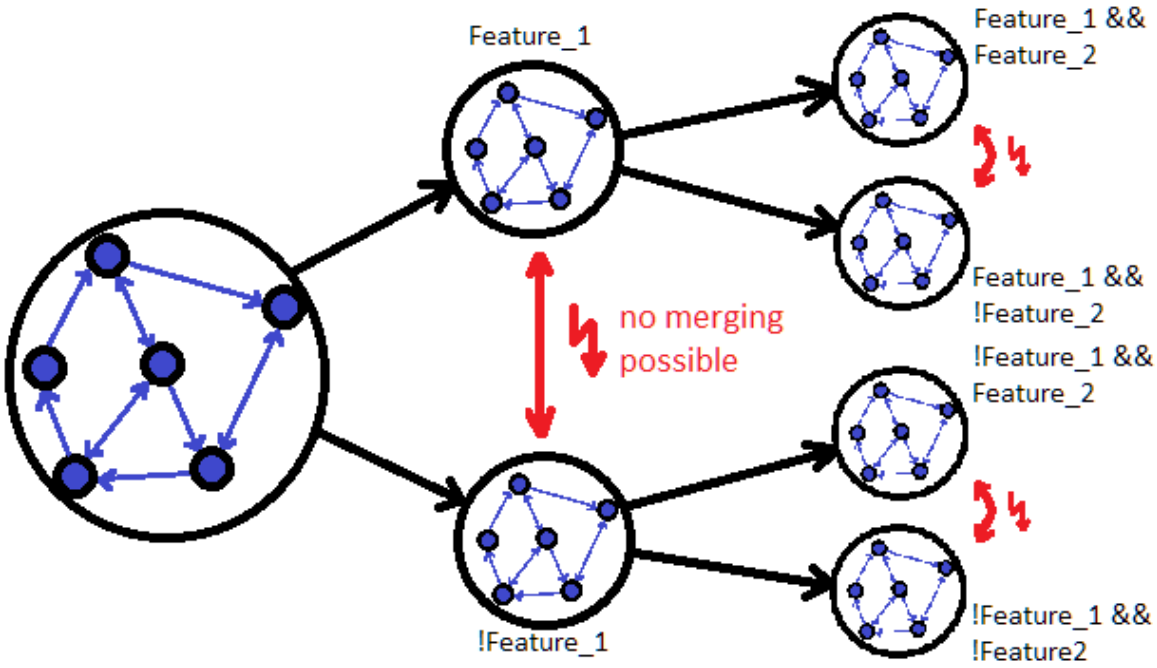


Figure 3.10: Merging problem with model checking of the metaproduct, with late initialization of feature selections.

selection values for features, at partial configurations. If a feature that is at a certain point unknown, at a partial configuration can only be set whether `true` or `false`, then this feature should be set, because if this feature is set at a later point, than it has another value and merging with the previous states is no longer possible, even though the configuration has not changed.

Model Checking of the Metaproduct with Runtime Assertions

With model checking applied to the described metaproduct, it is possible to analyze all valid products at once. Model checking was used in previous work for variability-aware testing [KvRE+12]. In our work, we additionally use design by contract, to benefit especially from blame assignment with runtime assertion checks. The work of Robby et al. provide a model checker that is able to check contracts [RRDH06]. However, they do not focus on a scope of product lines. In our work, we show how model checking in combination with runtime assertion checking can be applied to verify the metaproduct.

A main problem of model checking is that the analysis stops if an error is thrown, while it can be hard understand and localize this error, because it is not always obvious which method behaves incorrect to produce this error. This problem can be solved with design by contract. When executing a program with runtime assertions, contracts are checked additionally. If a contract is violated a signal is thrown. This additional information is useful, if an error is thrown afterward, but also if a test case fails, and even if something internal went wrong, but the test case did not fail.

When model checking is combined with runtime assertion checking it is not necessary to fully specify the whole program. Even if one method has simple clauses, design by contract is beneficial. This process has advantages, especially for the metaproduct, because of feature interactions it is not always obvious at which parts of the code and in which features something went wrong. Model checking with contracts can gain advantages of *blame assignment*, especially applied to the metaproduct.

When design by contract is applied to the metaproduct, we described contracts that must hold additionally. The check of the validity of the configuration (see Section 3.2.3), is already part of the variability encoding, so this check is unnecessary, because an invalid configuration cannot be reached. We also described a minimum feature selection for methods (see Section 3.2.3). This clauses can only be violated if a method is called wrong initially, meaning that the model was not set with the feature containing this method. It is also possible to detect invalid method call, (e.g. if a method does not exist at this configuration), but this can be done more efficiently be type checkers [AKGL10]. However, when checking the metaproduct with a model checker, the minimum selections for features are not necessary, but can be helpful to initialize the configuration.

Using model checking with runtime assertions has benefits, especially for error localization. However, we identified some possible problems with this approach applied to the metaproduct, in combination with late initialization of features. Because clauses of methods can depend on feature selections, but their values are not always given, so it is necessary to initialize them inside of the contract. Inside of contracts only `pure` methods are allowed, to be called, to ensure that contracts do not change the programs behavior. Depending on the JML compiler, this can be cause unexpected errors. This problem can be solved by initializing features used inside of a method, before calling the method. Listing 3.29 illustrates this solution. At the example, the contract of the method `update` depends on feature selections, which can be unknown. As a solution we renamed this method into `update$$` and called it within a new method `update`. The introduced method initializes all features the clause of the original `update` method contains. If the contracts are checked, then the features are already set with concrete values.

3.3.3 Comparison of Theorem Proving and Model Checking for Family-Based Verification

When using model checking to verify the metaproduct, it is not necessary to use specifications. Test cases can be executed for all valid program variants at once, while only the metaproduct has to be generated. Time wasting redundant calculations, especially of a core implementation that is equivalent in all program variants, are no longer necessary.

Model checking can profit from design by contract even if just a small set of methods is specified. However, model checking can often not be used to verify a program for all possible inputs, because of the state space explosion. Theorem proving can solve this problem. All possible inputs defined at the precondition for a method can be verified at once, while considering also the postcondition. It is often not easy to prove

Metaproduct

```
class Account {
    boolean update(int x) {
        FeatureModel.dailylimit();
        return update$$(x)
    }
    /*@ requires !FeatureModel.dailylimit ==> ... ;
      @ requires FeatureModel.dailylimit ==> ... ;
      @ ensures !FeatureModel.dailylimit ==> ... ;
      @ ensures FeatureModel.dailylimit ==> ... ;@*/
    private boolean update$$(int x) {
        if (FeatureModel.dailylimit) {
            update$$Base(x);
            return;
        }
        /* set withdraw */
        update$$Base(x);
    }
    /*@ ... @*/
    private boolean update$$Base(int x) {
        /* set balance */
    }
}
```

Listing 3.29: Metaproduct for model checking with contracts, avoiding initialization of features does not work it runtime assertions.

that a method fulfills its contract, depending on the power of the used tool and the complexity of the method. Nevertheless, if a method is finally proven it is no longer necessary to analyze the method. However, a proof can only show that a method fulfills its specification, or the method is called without violating the preconditions. Theorem proving is initially a time consuming task, but it is beneficial in safety-critical systems, because such systems must ensure to not harm human lives. Additionally, theorem proving can be beneficial applied to basic implementations, because such code is often reused, and other implementations are based on them, so they need to be correct.

For already specified product lines it can be a good approach to, apply an automated theorem prover to check which methods can already be proven. Nevertheless, theorem proving can only show the validity of methods. If a method does not fulfill its specification, a theorem prover is often not always able to show what causes this problem. Initially it is better to apply testing or model checking, to the program to find errors, because if these techniques can already find errors, in contrast to theorem proving it is easier to fix these problems, because of known input values and runtime assertions. If a method is finally fixed and testing or model checking do not find further errors, then theorem proving can be applied to finally prove the method. However, theorem proving should only be applied, if it is necessary and profitably to prove the program.

In Table 3.1 we compare different verification mechanisms applied to the metaproduct, which should help to select the mechanism for different purposes. The specification

effort of writing contracts for runtime assertion can be minimal, because even a single check can be helpful. Additionally, contracts for runtime assertions are usually of a simple kind, such as checks for null values. To use theorem proving for verification, the effort of writing specifications is much higher than for using them only for runtime assertions.

For testing and model checking it is necessary to write test cases (except if only the main method is used). The effort of writing test case, to find as many errors is usually much higher than for model checking. In general, testing has the lowest time for verification, while theorem proving does require a lot of time for verification.

Using the metaproduct, a test can only cover one product at once. However, we showed that with theorem proving and model checking it is possible to verify all products at once. Additionally, a test case can only execute a test with one input value, while model checking can cover a set of different inputs. However, theorem proving has no test case, but can verify methods for all possible inputs and program states, according to the precondition.

With testing and model checking, it is only possible to see if a test case fails for a given input value, except a error is thrown. With runtime assertion checking, the localization can be improved, because signals are thrown if a condition is violated. With theorem proving is possible to locate methods that cannot be proven, but it can be hard to understand and find the error.

With testing the rate of detected errors is low, but can be improved by using runtime assertions. With model checking the error detection rate increases, while this rate can be improved using runtime assertion checks, too. In general a theorem prover can find the most errors.

	Testing	Testing RAC	Model Checking	Model Checking RAC	Theorem Proving
Specifying	/	minimum	/	minimum	hard
Writing Test Cases	many	many	less	less	/
Verification Time	lowest	low	high	higher	highest
Verified Products	1	1	all	all	all
Verified Inputs	1	1	set	set	all
Error Localization	test case	method + test case	test case	method + test case	method
Error Detection Rate	lowest	low	high	higher	highest

Table 3.1: Comparsion of verification mechanisms applied to the metaproduct.

3.4 Summary

In this chapter, we discussed existing specification mechanism and decided that feature-based specification is the best solution for our purposes, because all products have specifications describing the properties of the product, while only feature modules need to be specified. Additionally, design by contract is possible at feature-based specification. With explicit contract refinement, features can be specified flexible, but with high expressiveness. With these techniques products can be generated with their own specifications. Verification of a certain product can be done efficient, but to verify the whole product line, it is necessary to verify more than one product. Product-based verification comes with many redundant calculations; especially core implementations are checked at any product.

Because generation and verification of all products or a subset thereof is a redundant or even impossible task, we proposed a family-based approach to verify all products at once. The variability of the software product line is converted into runtime variability represented as metaproduct that can simulate all products of the product line. The idea is that existing techniques and tools can be used to verify the metaproduct while considering the variability. Verification of the metaproduct has the same results as verification of all products, while nearly the complete time for generation and a lot of time for redundant checks is saved.

We discussed how verification tools can be applied to the metaproduct. We showed that the metaproduct can be used for testing, model checking and theorem proving, to verify the whole product line efficiently. Additionally, we discussed different benefits and disadvantages of these verification techniques. Because verification is a time consuming task, it is necessary to decide which verification technique is most beneficial. However, verification techniques should never be applied in isolation. Especially, theorem proving and model checking should always come with testing.

4. Tool Support in FeatureIDE

The last chapter described how a metaproduct for a software product line can be generated, while considering JML specifications. However, to create a metaproduct without tool support is a time consuming and error prone task.

Our goal is to provide a tool that can generate a metaproduct from feature modules with JML specifications using explicit contract refinement fully automatically. The tool should be able to generate the class `FeatureModel` out of an existing feature model. Additionally, existing verification tools must be able to handle the generated metaproduct without any customization.

Section 4.1 describes existing tool support for software-product-line development and generation of products. With FeatureIDE as platform for feature-oriented software development, FeatureHouse as generation tool for feature-oriented programming, and an existing extension of both supporting JML specifications, we have a good basis of existing tools for our goals. Section 4.2 describes our extension to generate a metaproduct based on FeatureIDE and FeatureHouse. Section 4.3 describes how existing verification tools, such as MonKeY and Java Pathfinder, can be applied to verify the metaproduct.

4.1 Contract Composition in FeatureIDE

FeatureIDE

FeatureIDE[1] is an open-source framework for Feature-Oriented Software Development (FOSD) available as an Eclipse[2] plug-in [TKB+13]. FeatureIDE supports all four phases of FOSD [TKB+13]:

Domain Analysis: The first phase focuses on feature modeling. FeatureIDE provides an editor to construct a feature model, which supports automated analysis of the feature model (e.g., identification dead features or redundant cross-tree constraints).

[1] http://www.fosd.de/fide
[2] http://www.eclipse.org/

Domain Implementation: This phase focuses on implementation of feature modules or code artifacts mapped to corresponding features.

Requirements Analysis: FeatureIDE provides an editor to configure the required system. The result is a selection of features or a configuration that is valid according to the feature model.

Software Generation: In the last phase, the actual product is generated based on the feature selection and the according feature modules.

The concept of FeatureIDE is to provide a platform for different composition mechanisms, while generation tools can easily be integrated and profit from existing support, provided editors, views and automatisms [TKB+13]. FeatureIDE supports several composition mechanisms. It provides support for *Feature-oriented* programing with the composition tools AHEAD for Java, FeatureC++ for C++ and FeatureHouse for different programming languages, such as C, C#, Java, Haskell, XML and JavaCC. FeatureIDE also supports the *preprocessors* Munge and Antenna for Java programs, DeltaJ for *delta-oriented* programming, and AspectJ for *aspect-oriented* programming.

FeatureIDE projects are extensions of existing Eclipse project structures [TKB+13]. For example, a Java project consists of a source folder and a binary folder. Files of the source folder are compiled into the binary folder. Also further commands for the compiler and additional libraries are possible. FeatureIDE does not inflict the given projects structure to use existing mechanisms provided by Eclipse. Because the files at the source folder are equivalent to a product, this folder is the path the composition tool generates the product out of feature modules, to use existing mechanisms of the underlying project, such as compilation. Therefor an additional folder is needed containing the feature modules[3].

To define the relations between features, a FeatureIDE project contains a file describing the feature model [TKB+13]. A FeatureIDE project can contain several configurations of which only one can be active at the same time [TKB+13]. This configuration is then used as input for the composition tool that composes the feature modules into the source folder. The result is a product representing the feature selection defined at the active configuration. Because the result is composed into the source folder of the underlying project, the project can be compiled, executed and analyzed as a normal project. With FeatureIDE it is also possible to generate all current configurations, or all products of the product line into separate projects (see Figure 4.1).

FeatureHouse

FeatureHouse[4] provides a language independent composition mechanism for feature-oriented programs [AKL13]. Based on a grammar for the given language, FeatureHouse

[3] An additional folder is not necessary for generation tools working directly at the source folder, such as Antenna and AspectJ.

[4] http://www.fosd.de/fh

is able to generate a feature structure tree out of a set of feature modules. This structure can then be composed into a product for the given feature selection. For superimposition it is necessary to implement mechanisms for composition of elements, such as methods and fields, in a feature-oriented manner (see Section 2.1.2).

In previous work, Benduhn extended FeatureHouse and FeatureIDE to support several JML contract-composition mechanisms [Ben12]. He altered the Java grammar of FeatureHouse to support JML keywords. Also the elements for method contracts and class invariants are integrated into the feature structure tree. Additionally, method contracts have a composition mechanism to support contract-aware feature composition. The extension provides different contract composition mechanisms, especially explicit contract refinement.

There are several variant managing tools, such as the commercial tool pure::variants[5], supporting modeling and configuration of products. There are also different feature-oriented programming languages, such as AHEAD [Bat06] and FeatureC++ [ALRS05]. However, only FeatureHouse supports contract-aware feature composition, what is a necessary basis for our approach. Additionally, there is no existing IDE support for all phases of feature-oriented software development, feature-oriented programming, and contract-aware feature composition, except of FeatureIDE with the integrated composition tool FeatureHouse.

The described tool support provided by FeatureIDE and FeatureHouse is a good basis to generate a metaproduct that supports JML specifications, out of a set of feature modules and the feature model automatically.

4.2 Metaproduct Generation

The default behavior of FeatureIDE is to build the active configuration into the source folder [TKB+13]. To build the metaproduct, we integrated an option to replace the standard mechanism with the mechanism for the metaproduct. Figure 4.1 shows a screenshot of FeatureIDE. At the menu of a FeatureIDE project with FeatureHouse as composition tool, metaproduct generation can be activated. After activation of metaproduct generation, the metaproduct is generated into the source folder, instead of a certain product.

At first, we generate the class `FeatureModel` of the metaproduct. For generation of the class `FeatureModel`, we use the internal representation of the feature model of the FeatureIDE project and Prop4J, a Java library extending SAT4J with arbitrary propositional formulas to calculate the CNF representation of the feature model for the method `valid` (see Section 3.2). With these information, we generate the class `FeatureModel` into the source folder.

To generate the metaproduct with FeatureHouse, the standard behavior must be customized. Instead of a configuration, FeatureHouse becomes a list of all concrete features

[5] http://www.pure-systems.com/pure_variants.49.0.html

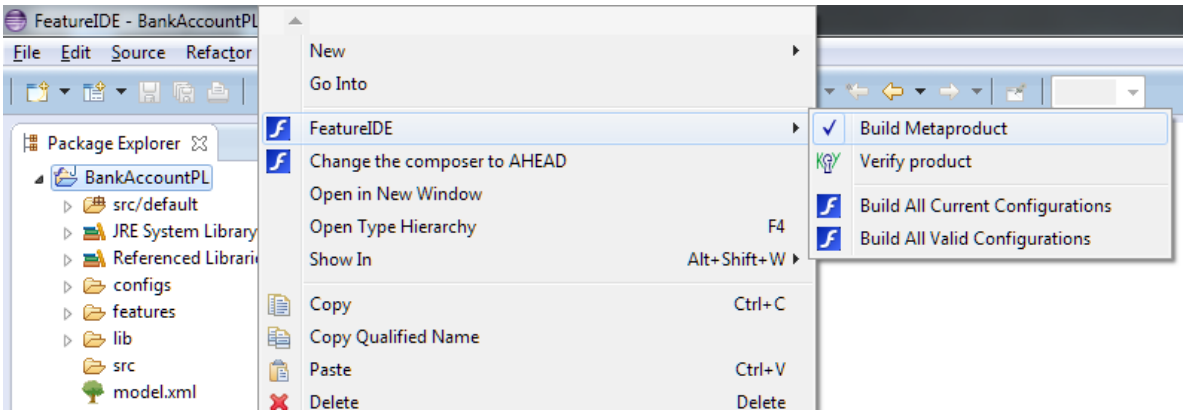


Figure 4.1: Activation of metaproduct generation, automated verification using MonKeY, generation of all products.

as input. To generate the metaproduct, the default composition mechanisms for Java and JML must be extended. To generate a metaproduct, we extended existing composition rules:

- the rule for method composition: `JavaMethodOverriding`
- the rule for field overriding: `FieldOverriding`
- the rule for constructor concatenation: `ConstructorConcatenation`
- the rule for contract composition: `ContractComposition`.

Because invariants are not composed by default, we integrated the composition rule `InvariantCompositionMeta` for invariants at the metaproduct. All generation and composition mechanisms for methods, constructors, fields, method contracts and invariants are described at Section 3.2.

To integrate metaproduct generation for Java using JML specifications, many default mechanism of FeatureHouse can be reused. Especially, parser, printer and the grammar for Java and JML do not have to be customized. However, we need to add the additional requires clauses (as described in Section 3.2.3), but not all method contract are composed. To add these clauses, we integrated a mechanism before composition. Because invariants are only introduced, we also used this mechanism to make invariants conditional.

If metaproduct generation for a FeatureIDE project using FeatureHouse as composition tool is activated, the metaproduct is built automatically into the source folder of the underlying Java project, to apply standard mechanisms such as compilers and verification tools without any further customization.

4.3 Verification of Metaproducts

The generated metaproduct can be used for analysis. In this section, we discuss different verification tools. First of all, standard analysis mechanisms especially provided by the compiler which are not part of this work, can be applied to the metaproduct. In Eclipse, warnings and errors are marked at the source files, while FeatureIDE is able to propagate these markers to the origin at the files of the feature modules [TKB+13]. To also profit by this mechanism, we adjusted the error propagation to also cover the metaproduct.

Model Checking with Java Pathfinder

Testing is very beneficial to cover a set of products, but to check all possible products it has its limits with a rising number of features and possible products. With model checking, it is possible to cover all execution paths at once. Applied to the metaproduct, we can check all valid configurations at once, while saving time wasted with redundant calculations.

We used the model checking tool Java Pathfinder[6] in combination with the testing tool JUnit[7]. With the described mechanism of Section 3.3.2 it is possible to efficiently cover and check all valid configurations.

As discussed in Section 3.3, model checking and theorem proving differ in some parts of the metaproduct. To generate a specific product for these different techniques we integrated a selection of the technique at the property page of the project. Figure 4.2 shows the part of the property page for FeatureIDE specific settings. At the combo box for metaproduct generation, it is possible to choose either theorem proving or model checking. This selection is then used to generate a metaproduct specific to the chosen verification technique.

Listing 4.1 shows a generated class `FeatureModel` for Java Pathfinder. The core feature *bankaccount* is initialized at a static block. With the method `Verify.getBoolean` provided by Java Pathfinder, the random feature selection can be simulated. For the method `valid`, we use Java Pathfinder instead of a solver, because of technical problems when using a solver in combination with Java Pathfinder. The implementation uses the structure of the feature model and checking random variables of unknown features. The method `Verify.ignoreIf` ignores the current path of the graph if the argument is `true`. As argument we use the formula of the model to check whether the selection is valid. Because the method `valid` should only check if a valid configuration is possible, the method should only generate one path. If a valid configuration is found, then the counter is incremented. If this counter is once incremented, then all further checks and states of the method `valid` are ignored, because the argument of the method `Verify.ignoreIf` is always `true`. If a partial configuration cannot lead to a valid configuration, then the partial configuration is ignored. At the metaproduct it is necessary to call the methods for features instead of the fields.

[6] http://babelfish.arc.nasa.gov/trac/jpf
[7] http://junit.org/

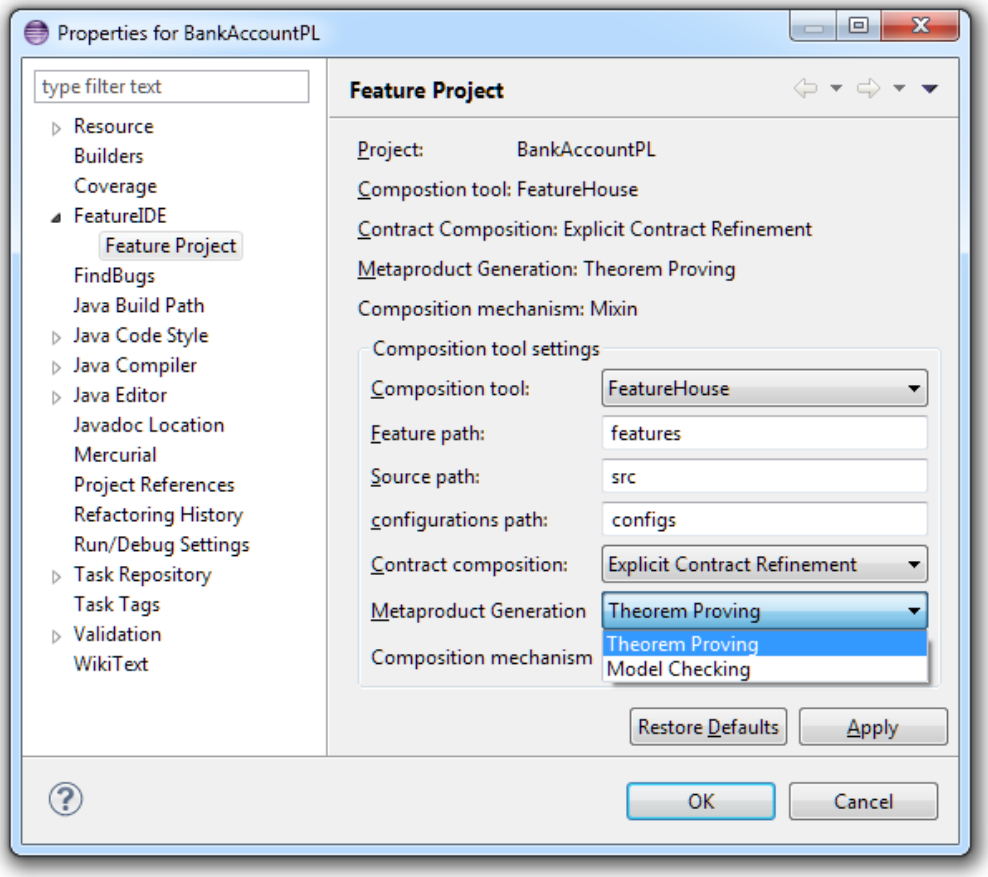


Figure 4.2: Selection of the metaproduct generation mechanism.

To use Java Pathfinder with JUnit[8], it is necessary to implement a class extending the class TestJPF provided by Java Pathfinder. Listing 4.2 illustrates such a test. To activate model checking instead of testing, the method `verifyNoPropertyViolation` needs to be called at the beginning of the test. To check feature specific properties, the field for the features can be initialized at the beginning of the test. The same can be done to only verify subset of all programs.

Theorem Proving with MonKeY

As theorem prover we use MonKeY[9], an extension of KeY. With MonKeY it is possible to parse and verify a Java project, specified in JML, fully automatically and interactive. MonKeY provides both mechanisms to handle contracts for verification, using the contracts instead of implementations and vice versa as discussed in Section 3.3.1. MonKeY proves each method with JML contracts. For a method, KeY proves that if a

[8] http://bluegulf.wordpress.com/2011/04/20/tutorial-using-java-pathfinder-together-with-junit-in-eclipse-environment-setup/ A helpful tutorial how to setup JPF to use it together with JUnit

[9] http://www.key-project.org/

```
public class FeatureModel {
    public static Boolean dailylimit, bankaccount;
    static {
        bankaccount = true;
    }
    public /*@pure@*/ static boolean valid() {
        Verify.resetCounter(0);
        boolean bankaccount_ = true;
        boolean dailylimit_ = dailylimit != null ? dailylimit : random();
        Verify.ignoreIf(Verify.getCounter(0) != 0
            || !(!dailylimit_  ||  bankaccount_));
        Verify.incrementCounter(0);
        return true;
    }
    private static boolean random() {
         return Verify.getBoolean();
    }
    public static boolean bankaccount() {
        return bankaccount;
    }
    public static boolean dailylimit() {
        if (dailylimit == null) {
            dailylimit = random();
            valid();
        }
        return dailylimit;
    }
}
```

Listing 4.1: Generated class `FeatureModel` for Java Pathfinder.

method is called without violation of its preconditions, then the method executes without throwing any error and also fulfills its postcondition when it has finished. Listing 4.3 illustrates the proof as formal description.

Additionally, we integrated MonKeY to be used in FeatureIDE. With a menu entry (see Figure 4.1), MonKeY proves the selected project automatically. The methods which cannot be proven are marked at the generated product. These markers are then automatically propagated to the origin at the corresponding feature module. All generation mechanisms for MonKeY are implemented as described in Section 3.2.

The more often a method is refined, the more complicated the resulting contract seems to be, which can cause the prover to not be able to verify the contract, also if the method can be proven in all products. However, after a closer look we see that the contract is structured in alternative proves, because of the alternative implications (see Section 3.2.3). One proof must hold if the corresponding feature is selected, while the other proof is then always true, and vice versa. The contract is growing, but this causes only more steps to be proven. At our observations the theorem prover needs more steps

```
public class AccountTest extends TestJPF {
    @Test
    public void updateTest() {
        if (verifyNoPropertyViolation()) {
            Account a = new Account();
            a.update(100);
            assertEquals(100, a.balance);
        }
    }

    @Test
    public void updateDailyLimitTest() {
        if (verifyNoPropertyViolation()) {
            FeatureModel.dailylimit = true;
            Account a = new Account();
            a.update(10000);
            assertTrue(a.update(-900));
            assertFalse(a.update(-200));
        }
    }
}
```

Listing 4.2: Example tests for Java Pathfinder and JUnit.

```
precondition ->
<try {
    exc = null;
    /* method */
} catch (Exception e) {
    exc = e;
}>
-> exc == null && postcondition
```

Listing 4.3: Formal description of a proof with KeY.

for the complete proof, but if the corresponding method can be proven at all products, then it can also be proven at the metaproduct.

4.4 Summary

In this chapter, we presented our tool support to automatically generate a metaproduct. Based on the existing tools FeatureIDE and FeatureHouse we had a good basis to develop our tool. We showed how existing verification tools can be reused without customization to analyze a metaproduct. Thereby the metaproduct can simulate certain products or all products of the product line at once. JML specifications are not necessary for testing and model checking, but these techniques can profit from JML specifications, especially for error localization.

With metaproduct verification, it is possible to verify a whole software product line at once, without a need to generate all products. However, in contrast to previous work

which focused only on specification of feature modules [Ben12, TSK+12], we showed how generated products with JML specifications can be verified.

This thesis based on the previous work of Thüm et al. [TSAH12]. With an automatic mechanism, we were able to adjust the proposed mechanisms of metaproduct-generation, because test and evaluation of the metaproduct was possible in a larger scale than before.

Our tool support is available with FeatureIDE 2.6.5. The case study BankAccount is available at SPL2go[10], a repository for case studies of software product lines.

[10] http://spl2go.cs.ovgu.de/

5. Evaluation

We described our tool support to automatically generate a metaproduct and how existing verification tools can be applied to it for family-based verification. For evaluation, we use and extende an existing case study of a bank accout product line [TSAH12]. In this chapter, we present and discuss our results.

5.1 Case Study BankAccount

In previous chapters, we used excerpts of the bank account product line for illustration. Figure 5.1 shows the complete feature model of the bank account product line. It consists of eight features and a total of 72 different program variants. The product line is implemented using feature-oriented programming with the composition tool FeatureHouse. All methods have JML specifications based on explicit contract refinement as a contract composition mechanism.

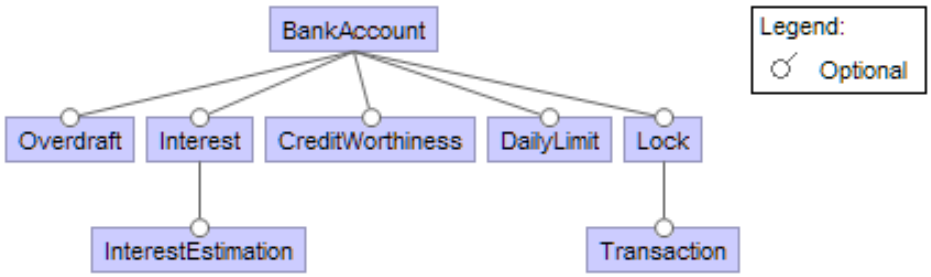


Figure 5.1: Complete feature model for the bank-account product line.

For statements about a trend for a growing variability of the product line, all statistics are of the same kind. We start with a simple product line consisting of only one feature. Then, we add a feature to the product line in every step. For every step, we compare the verification time of all products available in the current step, with the metaproduct.

For product-based verification, we expect an exponential growing for verification time. For family-based verification, we expect polynomial growing for verification time with respect to the number of features.

Table 5.1 shows the order in which the features are added to the product line. The third column shows the current number of features of the product line, and the right column the number of program variants. The order has two rules. The root feature has to come first, and features implying other features such as *InterestEstimation* and *Transaction*, need to be added after their parent feature. The order of the other features does not follow any intentional rules. For evaluation we used desktop PC with Intel Core i5-2500 CPU @ 3.30GHz and 8 GB DDR 3 RAM.

Step	Added Feature	#Features	#Products
1	BankAccount	1	1
2	Overdraft	2	2
3	Interest	3	4
4	InerestEstimation	4	6
5	CreditWorthiness	5	12
6	DailyLimit	6	24
7	Lock	7	48
8	Transaction	8	72

Table 5.1: Insertion order of features into the product line for evaluation.

5.2 Time for Metaproduct Generation

First, we compare the time to generate all products with the generation of the corresponding metaproduct. We only compare the times for parsing, composition and printing. For generation of all products also the calculation for valid configurations can have impacts on the result, but this highly depends on the algorithm for calculation and, thus is ignored in our case study. If the products are generated into separate Eclipse projects, then this has also an impact on generation. However, we only compare times for generation with FeatureHouse, to make expressive and comparable statements about times for generation.

Figure 5.2 shows the results for product-based generation compared to the generation of the metaproduct. For the product line with 8 features and 72 program variants family-based generation saves about 96% of the time compared to generation of all products.

Generation of all products highly depends on the number of valid products, with an exponential expenditure of $\mathcal{O}(2^n)$. In contrast, the generation of the metaproduct needs

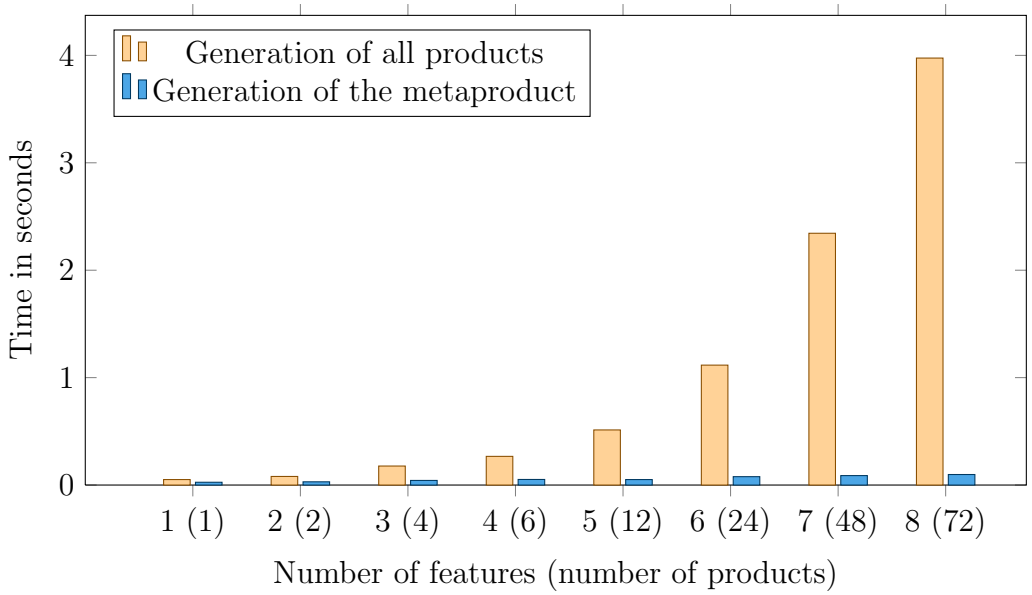


Figure 5.2: Time needed for generation of the metaproduct compared to generation of all products.

only slightly more time than the generation of a product that contains all features. Time for generation of the metaproduct took about 86 ms and generation of the full product about 83 ms. However, the time for generation varies a lot, because of the small times. The metaproduct generation only depends on the number of features and the size of the feature modules.

In general, the generation of a certain program needs less time than verification. However, with an increasing number of products and size of the feature modules, even generation of all products becomes impractical. Especially because other factors have impacts, such as limited resources, error propagation, and handling of all products.

5.3 Verification of the Metaproduct Using Theorem Proving

Automated theorem proving using MonKeY consists of two steps. The first step is to type check the specified code and to generate the proof obligations. In the second step, the proof obligations are verified automatically. Because the first step is magnitudes less time consuming than the proof itself, we do not distinguish between these two steps. The first step is not optional anyhow, thus it is part of the verification process.

Figure 5.3 shows the results for automated theorem proving using MonKeY for product-based verification compared to family-based verification. The time for verification of each product is less than verification of the metaproduct. However, if all products need to be verified, then there are redundant proofs, which increase the time for verification of all products by the number of products.

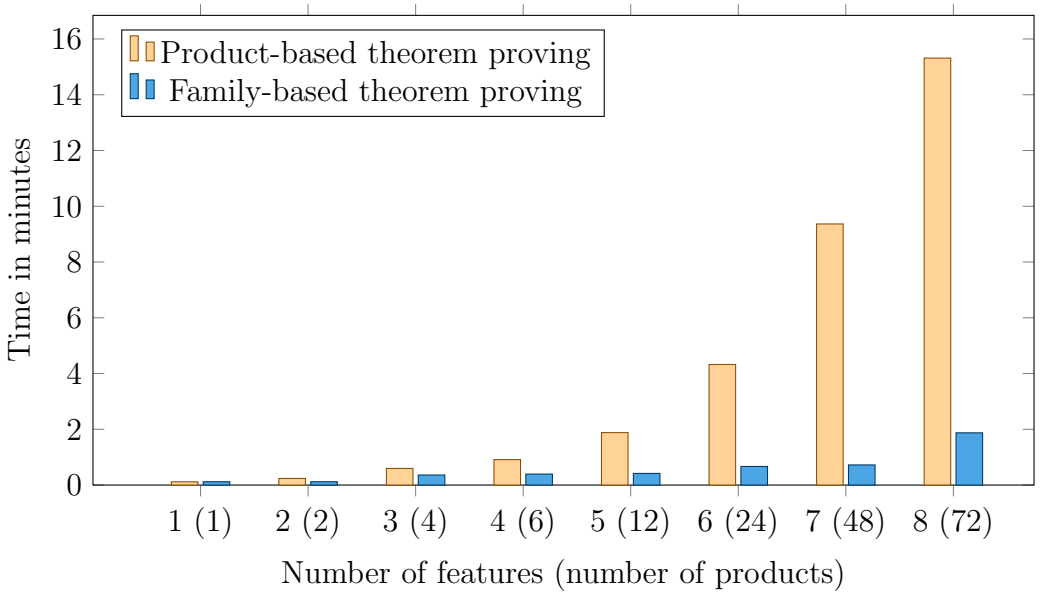


Figure 5.3: Time need for family-based theorem proving compared to product-based theorem proving.

First, we look at the step from 6 to 7 features. At this step, the feature *Lock* is introduced into the product line. This feature does not inflict the other methods directly and also does not refine any methods. The complexity of the metaproduct only increases slightly, while the number of products doubles. So the time for product-based verification also doubles, while the verification of the metaproduct only increases slightly by the proofs for the feature *Lock*.

The time for verification of the metaproduct doubles, at the step from 7 to 8 features, where the feature *Transaction* is introduced. The implementations of the feature *Transaction* interacts with 6 of 8 features. Additionally, the complexity of the methods and it's contracts are higher than in all other features. To prove the method `transaction` in the metaproduct least about 63 seconds, which is more than half of the complete time for verification of the metaproduct. The more features are inflicted by a method, the higher is the time to prove the method at the metaproduct. However, at product-based verification the method `transaction` needs to be verified 24 times. Because at product-based verification many proofs are done redundantly, family-based analysis is significantly more efficient, then to verify all products. In our case study, family-based theorem proving saved 87% of verification time compared to verification of all products, for the product line with 72 products. Verification of the metaproduct took about 112 seconds and verification of all product about 918 seconds.

5.4 Verification of the Metaproduct Using Model Checking

For model checking, we used a set of different tools integrated into Eclipse. For execution of the test cases we used the testing tool JUnit. As a model checker, we used the Java Pathfinder. We only used the standard version of Java Pathfinder. This tool is highly customizable and provides different variants, but for our case study the standard version is sufficient.

For evaluation, we created several test cases provided by each feature, so the generated products only contain the test cases of the corresponding feature selection. For the metaproduct the corresponding feature of the test case must be set before executing the test case. Listing 5.1 illustrates such a test case. The test is provided by the feature *DailyLimit* and is contained in the corresponding feature module. The feature *DailyLimit* limits the maximum amount of withdrawn money to 1000, so the test can only be valid at products with the feature *DailyLimit*. To check the metaproduct the feature *DailyLimit* has to be set.

```
@Test
public void updateTestDailyLimit() {
    if (verifyNoPropertyViolation()) {
        /* only initialize for the metaproduct */
          FeatureModel.dailylimit = true;

        Account a = new Account();
        a.update(10000);
        assertTrue(a.update(-900));
        assertFalse(a.update(-200));
    }
}
```

Listing 5.1: Example test case for the feature *DailyLimit*

Model Checking without Runtime Assertions

First, we show how model checking can be applied without considering contracts. Figure 5.4 shows the time for model checking all products compared to the family-based approach, whereas the assertions are not checked. For the metaproduct, the time increases slightly, depending on the number of affected features and complexity of the test cases. For the product-bases approach the time increases exponentially with the number features.

In the step from 7 to 8 features, where the feature *transaction* is introduced, grows different than in all other steps, because the test case for the feature *Transaction* is more complex. Transactions are done in parallel threads, to check all interleaves between different transactions over a set of accounts. A simple test can execute the test case with no additional effort, but to find possible errors with simple testing is than hardly possible.

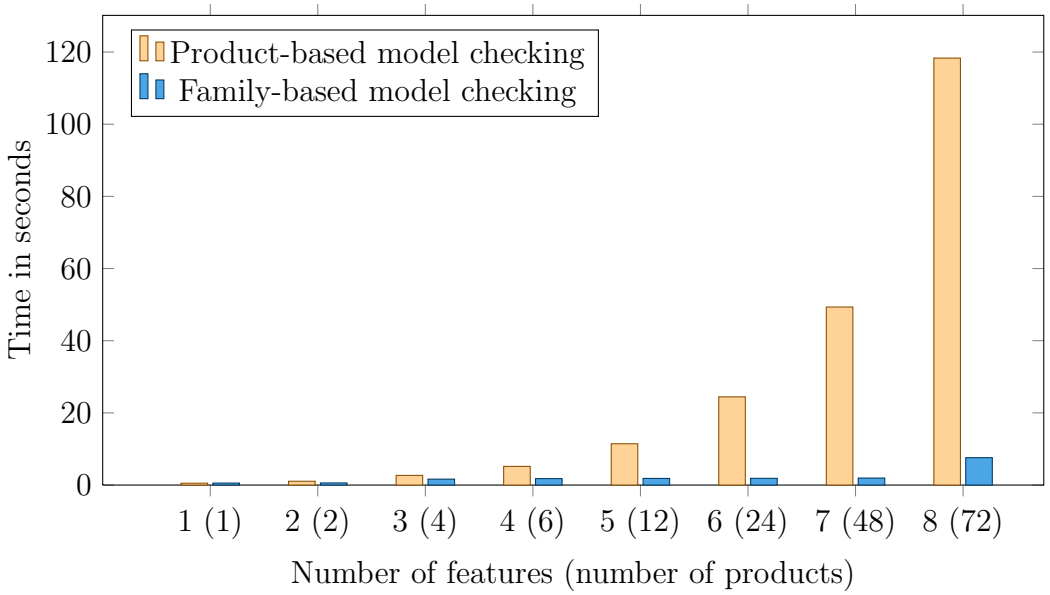


Figure 5.4: Time need for family-based model checking compared to product-based model checking, without runtime assertion checks and with parallel transactions.

The Java Pathfinder can handle the problem of thread interleaving, with higher effort in case of a larger state space, but all possible thread execution orders can be checked at once. At the metaproduct all program variants are checked in addition. In that case, we found errors with synchronization and locking of transactions, which was successfully verified by theorem proving, but the specification and prove have not covered transactions in parallel threads, because KeY does not support multithreading.

Verification of the metaproduct saved about 93% of time compared to verification of all products for the complete product line with 72 products. Model checking of the metaproduct without runtime assertions took about 7.6 seconds and 118 seconds for all products.

Model Checking with Runtime Assertions

We used the Eclipse plug-in OpenJML[1] to generate runtime assertions. OpenJML is a tool set for JML and provides a JML runtime assertion compiler. This compiler generates runtime assertions in form of console outputs, in contrast to throwing errors, which would cause the model checker to stop at this point.

Figure 5.5 shows the results for model checking with runtime assertion checks. Transaction within threads was not possible, because model checking has not worked for threads on the code with runtime assertions, so transactions are just executed serially. Model checking with runtime assertion checks has also not worked for products, because the Java Pathfinder and the JML compiler seem to interact.

[1] http://jmlspecs.sourceforge.net/

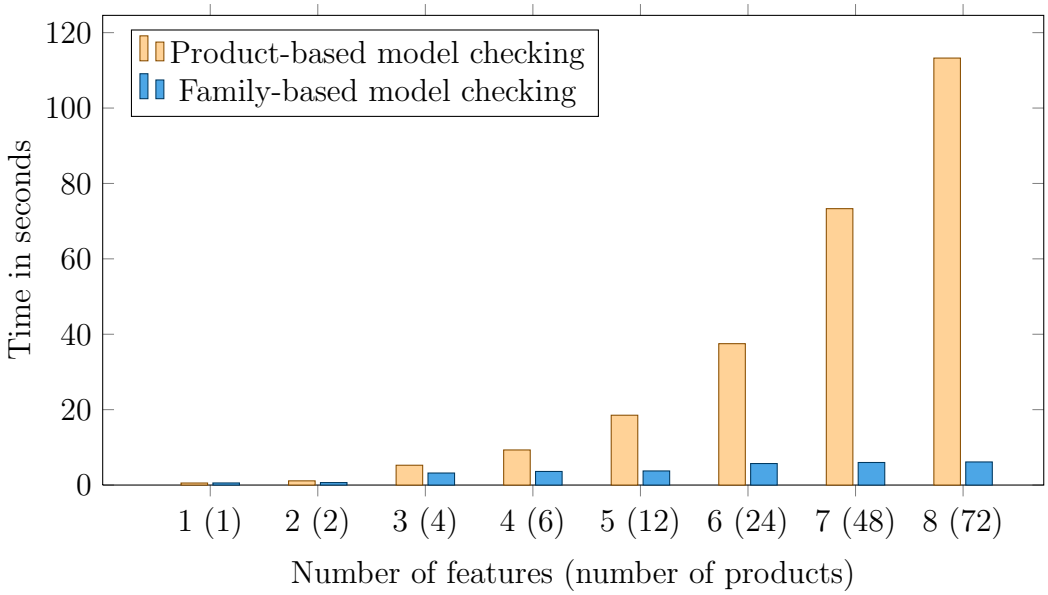


Figure 5.5: Time need for family-based model checking compared to product-based model checking, with runtime assertion checks, and without parallel transactions.

All checks last longer compared to the checks without runtime assertions, except of the transactions. Family-based model checking with runtime assertions saved up to 94% at the complete product line compared to the product-based approach. However, using runtime assertion checks was useful, especially to find wrong initialized features at test cases of the metaproduct. Model checking of the metaproduct with runtime assertions took about 6.1 seconds and 113 seconds for all products.

5.5 Summary

We were able to apply model checking and theorem proving to the metaproduct. Our family-based approach has worked successfully at the case study. Product-based verification has not found further errors. With the family-based approach the exponential complexity of software product lines is reduced to polynomial complexity. With the metaproduct we saved 97% of time for generation, 87% for theorem proving, 93% for model checking without runtime assertions, and 94% for model checking with runtime assertion checks.

After theorem proving we still found errors with model checking, because the theorem prover has not considered parallel executions. After and while fixing the method `transaction`, theorem proving of the method was helpful, to ensure that the changed to fix the implementation did not cause new errors.

6. Related Work

This thesis focuses on software product lines using feature-oriented programming for generation, explicit contract refinement as contract-composition technique and a family-based analysis for verification. In this chapter we give an overview on related work on the field of product line verification.

The work of Thüm et al. [TSK+12] and Benduhn [Ben12] showed how design by contract can be applied to feature-oriented programming. Based on their prior work we applied verification tools to analyze products and the metaproduct.

Generation

Because the potential number of products of a software product line explodes with an increasing number of variability, other work focused on products, but only a subset of all products. The idea is to find as many errors as possible with this subset of product. So, there is a need for a good strategy to select as less products as necessary, while finding as many errors as possible. Tartler et al. worked on code coverage [TLD+12]. Because there is usually no configuration where any code is contained in the resulting product (e.g., at alternative statements of preprocessors, or alternative features), there is a need to select as less products as possible where all code is at least once at one product. Other strategies focus on feature interactions, based on the assumption that most errors are cause by feature interaction. Oster et al. used pair-wise testing [OMR10], while Perrouni et al. used t-wise generation to reduce the number of products [PSK+10].

Type Checking

In this work, we assume a type-save product line. However, type safety must be guaranteed. Hence, other work proposed techniques and type systems to efficiently analyze the product line to be type-save. *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the*

kinds of values they compute [Pie02]. Typically type systems are known form compilers where type safety is checked automatically [Pie02]. As mentioned in Section 3.1 the metaproduct should not be used for type checking.

In software product lines it is not useful to type check all possible programs separately. The metaproduct cannot find all compiler errors because all fields are exist. To find invalid method calls to not existing methods is possible with our introduced requires clauses for minimum feature selection, but type checking the product line can be do this more efficient. Kästner et al. [KA08] gives a formal approach on how to guarantee that all possible variants of a software product line are well typed, by extending the Featurewight Java's type system. Apel el al. [AKGL10] developed a type system for feature oriented product lines. Schaefer et al. proposed compositional type-checking for delta-oriented programing by presenting a minimal core calculus [SBD11]. With a constraint-based type system they analyze delta modules in isolation, and are able to reuse analysis. They are able to establish that all products are well-typed by combining analysis results.

Model Checking

At this work we used runtime assertions in combination with model checking. However, we also discussed possible disadvantages, especially with splitting inside of assertions. Another way is to check contracts is to extend a model checker to directly check contracts [RRDH06].

Apel et al. developed the tool SPLverifier [ASW+11]. They used a metaproduct to detect feature interactions with the model checker CPAchecker.

As discussed in Section 3.3.2 merging between states of different products is not possible, because the values of fields for features differ. Kästner et al. used a metaproduct in combination with their extension jpf-bdd of the Java Pathfinder [KvRE+12], for model checking the software product line. They give a solution of the merging problem by introducing binary decision diagrams to merge states, where features have different selection states.

Theorem Proving

In this work we proved the metaproduct to prove all products at once, to avoid redundant verification. An alternative solution is proposed by Bruns et al. [BKS11]. Initially they proved a certain product. To reduce redundant calculation they used this proof to only prove the differences to the proof of the next product. Thüm et al. [TSKA11] proposed proof composition. They write partial proofs for each feature. These partial proofs are composed to retrieve a proof for a certain product.

Summary

Verification of software product lines is a wide field of current research. The main goal is to handle the exponential number of possible product. There are different analysis techniques to achieve this goal, such as optimized product-based, feature-based, family-based, and combination thereof [TAK+12].

7. Conclusion

Verification of software is a major part of the software development process. Reliability is important for software artifacts that are reused in different kinds of applications, and for safety-critical systems. To increase the correctness and robustness of software, the specification concept of design by contract is beneficial. Several programming languages or extensions thereof such as JML already support design by contract. With specifications, it is possible to increase and even prove the correctness of a program.

The idea of software product lines is to develop different customized products that share a common code base. This comes with many advantages for development costs, because there is no need to develop single systems from scratch. With high reuse, the number of potential bugs can be reduced, in case of known artifacts. However, also new bugs can be introduced by interactions of these artifacts. Single systems or a set of possible systems of a product line can be verified efficiently in isolation. However, high customization comes with major disadvantages. A product line with n optional, independent features consists of 2^n distinct products. To handle such a high number of products comes with difficulties and cannot be done efficiently, especially if user interaction is needed. To compensate the problem of a exponential number of possible products, different strategies such as t-wise verification or code coverage come with the compromise, to only verify a representative set of products to cover as many errors as possible. However, these strategies cannot verify all products of the software product line.

In this work, we used a family-based approach to solve the problem of an exponential number of products of a software product line. With family-based verification, it is possible to verify all products line at once without a need to generate all products, while redundant calculations are reduced. We used a metaproduct that can simulate all products, while it contains the code artifacts of all features and simulates the products by means of runtime variability. Additionally, we introduced runtime variability into contracts, so the specification always matches the simulated product. To use the

metaproduct for verification, tool support is needed. As part of this work we provide an extension of FeatureIDE and FeatureHouse to automatically generate a metaproduct with JML specifications.

For evaluation of our concept we used a software product line with 8 features and 72 products. We successfully applied the model checker Java Pathfinder and the theorem prover MonKeY to verify the whole product line. The verification of the metaproduct was able to find the same errors as verification of all products. However, with the family-based approach we saved up-to 96% generation time and up-to 94% of verification time.

When model checking is applied to the metaproduct, features without an effect on the test case are simply ignored and many redundant calculations of all products are saved, especially for the core implementation. Model checking the metaproduct can be costly depending on the number of features. However, the test must still be executed at all products.

With the theorem prover MonKeY, we showed that proving the complete software product line using the metaproduct can be done efficiently. Verification of a method that affects many different features is a time consuming task, but must only be done once. In contrast, methods without variability can be verified efficiently, while the method has to be verified redundantly in all products. The less features are affected by a proof of a method; the more efficient is verification of this method at the metaproduct compared to proving the method at all products.

For larger product lines with more variability, we expect even a larger speed-up. Complicated checks including many different features are more expensive, but the exponential cost of product-based verification does always compensate this with redundant checks. At even larger product lines where it is practically impossible to generate all products, our approach should be able to verify the whole product line anyhow.

8. Future Work

Future work should investigate the correctness of our proposed metaproduct. Currently, we generated the metaproduct and verified it with theorem proving and model checking. We fixed all with metaproduct verification found errors. After there were no errors left we generated and verified all products. With product-based verification we were not able to find additional errors. However, it is not guaranteed that metaproduct verification can find all errors that product-based verification can find. To ensure that no error is missed, a formal correctness proof for our metaproduct generation is necessary. Therefore it is necessary to prove that the metaproduct behaves exactly as the simulated product for every configuration. After this proof it is no longer necessary to check products. Next to a formal proof, other case studies are needed to investigate gaps at our current approach. For meaningful conclusions more representative product lines are needed, with more complex code, a higher number of features, and interactions between them. A major question is, if verification of the metaproduct is possible for large product lines, especially for product lines where it is practically impossible to generate all products.

For theorem proving, we discussed the technique *using contracts* (see Section 3.3.1). We did not include this technique into our evaluation, because proofs for this technique are hard to close. However, we discussed major benefits with *using contracts* applied to the metaproduct. Future work should show how *using contracts* can speed-up metaproduct verification compared to *method inlining*, especially if only program parts need to be proven (e.g., in case of software evolution). Currently theorem proving needs to be done from scratch. However, during software evolution only program parts or single methods need to be verified. Future work should only prove the program parts that need to be proven. Using existing proofs, and only verify methods that have changed, can have high speed-ups for theorem proving compared to other verification techniques, such as model checking and testing.

For model checking we used the standard Java Pathfinder, which is suitable for our purposes. However, there are several extensions of the Java Pathfinder for different purposes, such as the extension *jpf-bdd* [KvRE+12, vRAR11]. As discussed merging between states of different feature selections is not possible. *jpf-bdd* focused on solving this problem with annotated fields and binary decision diagrams. However, future work should investigate how suitable *jpf-bdd* is, applied our proposed metaproduct.

As discussed in Section 3.3.2 there are separate groups of states for different feature selections. Because merging between these states is not possible, it can be an advantage to remove already verified state groups of some feature selections, to save memory, because these states cannot be reached anymore.

In this work, we only applied model checking with runtime assertion checks and theorem proving. It is also interesting how *static analysis* tools can handle the metaproduct with its runtime variability and variable contract. Especially, because static analysis is less time consuming then model checking or theorem proving, but also a very powerful analysis techniques. Another question is, how *test-case-generation* tools for specified programs can handle the metaproduct. What kinds of test cases are generated, and are they equivalent to the test cases generated by each product?

This work only focused on basic JML constructs. However, JML provides additional keywords (e.g., `also` for alternative contracts). How are these constructs already supported, or how does our approach need to be adjusted to support them?

We used family-based verification for analysis of all products at once. Thüm et al. [TAK+12] proposed feature-based verification to analyze feature modules in isolation. Feature-based analysis can be combined with family-based analysis as *feature-family-based* analysis [TAK+12], to profit from both techniques. However, future work needs to investigate possible benefits of combining both techniques, especially because independent parts at the feature modules are also independent at the metaproduct.

For product-based verification we generated all products. However, there are several other product-based techniques, such as t-wise verification as discussed in Section 3.1. Future work needs to investigate, in which cases it is beneficial to use either a product-based approach or to verify the metaproduct, and if there is a point on which family-based verification is faster than a product-based approach (e.g., pair-wise verification). Future work should use *mutation analysis* to compare effectiveness and efficiency of different analysis techniques and approaches.

Bibliography

[AKGL10] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010. (cited on Page 34, 54, and 78)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79, January 2013. (cited on Page 4, 9, 32, and 60)

[AL08] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l Symposium Software Composition (SC)*, pages 20–35, 2008. (cited on Page 8)

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 125–140, Berlin, Heidelberg, New York, London, 2005. Springer. (cited on Page 61)

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM. (cited on Page 1)

[ASW+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375, Washington, DC, USA, 2011. IEEE. (cited on Page 30 and 78)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20, Berlin, Heidelberg, New York, London, 2005. Springer. (cited on Page 7)

[Bat06] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Proc. Generative and Transformational Techniques in Software Engineering*, pages 3–35, Berlin, Heidelberg, New York, London, 2006. Springer. (cited on Page 61)

[Ben12] Fabian Benduhn. Contract-Aware Feature Composition. Bachelor's thesis, University of Magdeburg, Germany, 2012. (cited on Page 2, 24, 25, 61, 67, and 77)

[BFL+11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Comm. ACM*, 54:81–91, June 2011. (cited on Page 1)

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, Berlin, Heidelberg, New York, London, 2007. (cited on Page 46 and 47)

[BKS11] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *LNCS*, pages 61–75, Berlin, Heidelberg, New York, London, 2011. Springer. (cited on Page 78)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA, 2000. (cited on Page 8)

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachussetts, 1999. (cited on Page 16, 17, and 18)

[Cla08] Edmund M. Clarke. 25 Years of Model Checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008. (cited on Page 16)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 2 and 5)

[CR06] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006. (cited on Page 16)

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008. (cited on Page 16)

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 258–267, Washington, DC, USA, 1996. IEEE. (cited on Page 24)

[Fre79] Gottlob Frege. *Begriffsschrift, Eine der arithmetischen nachgebildete Formalsprache des reinen Denkens.* Halle(S), Jena, Germany, 1879. (cited on Page 18)

[Hoa03] C. A. R. Hoare. Assertions: A Personal Perspective. *IEEE Ann. Hist. Comput.*, 25(2):14–25, April 2003. (cited on Page 16)

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, January 1997. (cited on Page 1 and 2)

[KA08] Christian Kästner and Sven Apel. Type-Checking Software Product Lines—A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267, Washington, DC, USA, 2008. IEEE. (cited on Page 34 and 78)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 2 and 6)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, Berlin, Heidelberg, New York, London, 1997. Springer. (cited on Page 9)

[KvRE+12] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8, New York, NY, USA, SEP 2012. ACM. (cited on Page 3, 28, 50, 53, 78, and 82)

[LC06] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, September 2006. (cited on Page 1 and 13)

[Mey92] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992. (cited on Page 1 and 13)

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 196–210,

Berlin, Heidelberg, New York, London, 2010. Springer. (cited on Page 27 and 77)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, New York, London, September 2005. (cited on Page 2, 5, and 6)

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, USA, 2002. (cited on Page 78)

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443, Berlin, Heidelberg, New York, London, 1997. Springer. (cited on Page 2 and 9)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 459–468, Washington, DC, USA, April 2010. IEEE. (cited on Page 27 and 77)

[RRDH06] Robby, Edwin Rodriguez, Matthew B. Dwyer, and John Hatcliff. Checking JML Specifications Using an Extensible Software Model Checking Framework. *International Journal on Software Tools for Technology Transfer*, 8(3):280–299, 2006. (cited on Page 53 and 78)

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 77–91, Berlin, Heidelberg, New York, London, 2010. Springer. (cited on Page 9)

[SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional Type-Checking for Delta-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56, New York, NY, USA, 2011. ACM. (cited on Page 78)

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, April 2012. (cited on Page 12, 19, 20, 21, 22, 25, 27, 28, 78, and 82)

[TKB+13] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 2013. To appear. (cited on Page 4, 27, 59, 60, 61, and 63)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 191–200, Washington, DC, USA, August 2011. IEEE. (cited on Page 7)

[TLD+12] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, January 2012. (cited on Page 77)

[TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA, September 2012. ACM. (cited on Page 3, 28, 30, 37, 67, and 69)

[TSK+12] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 255–269, Berlin, Heidelberg, New York, London, March 2012. Springer. (cited on Page 2, 19, 20, 21, 24, 25, 67, and 77)

[TSKA11] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, DC, USA, March 2011. IEEE. (cited on Page 78)

[Tur49] Alan Turing. Checking a Large Routine. In *Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949. (cited on Page 13)

[vRAR11] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. *Proc. Java Pathfinder Workshop*, 2011. (cited on Page 82)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den April 22, 2013