University of Magdeburg

School of Computer Science

Master's Thesis

Product-Line Verification with Abstract Contracts

Author:

Stefan Krüger

December 15, 2014

Advisors:

Prof. Gunter Saake
Dipl.-Inform. Thomas Thüm
Department of Technical and Business Information Systems

Dr. Richard Bubel
Department of Computer Science

Krüger, Stefan:
Product-Line Verification with Abstract Contracts
Master's Thesis, University of Magdeburg, 2014.

Abstract

Software product lines are used for highly efficient development of software products with a common code base. As they are used increasingly often in safety-critical systems, means of verification have come into focus of research, but efficient verifications of software product lines are still a challenge. To verify a software product lines all its products need to be verified. Different approaches have emerged that try to accomplish this. In this thesis, we revisit the discussion about benefits and weaknesses of the different approaches, with a focus on product line evolution. In product-based verification approaches, all products are generated and verified individually, which becomes infeasible with an increasing number of products. To solve this issue, in family-based theorem proving, a single metaproduct is generated that incorporates the variability of the whole product line. This approach, however, may cause problems because even for small changes in a single feature, the verification must be repeated completely. Feature-based approaches address this issue by verifying features in isolation, but as features regularly interact with each other, solely feature-based cannot be realized.

In order to solve the problems the individual approaches have, we propose a new approach combining feature- and family-based theorem proving. In our approach, we separate the verification in two phases. In the feature-based phase, feature modules are transformed into feature stubs, which are then verified. The proofs are saved for reuse. To realize feature module dependencies, abstract contracts are used to serve as placeholders in the verification of the feature stubs. In the family-based phase, a metaproduct is generated that is structured, so that the partial proofs obtained in the feature-based phase can be reused. When a once verified product line evolves, the feature-based verification needs to be repeated only for features with changes. Furthermore, we provide tool support for the feature-based verification phase for FeatureHouse Projects specified with JML.

To evaluate our approach, we compare it with four completely family-based approaches regarding overall proof complexity and reuse potential. We find, that our approach can reduce the proof complexity up to 65% compared to a family-based approach. Furthermore, we are able to reuse about 8.6% of an original full proof of a product line.

Acknowledgements

I would like to take the opportunity to thank my advisors Thomas Thüm, Dr. Richard Bubel, and Prof. Gunter Saake for giving me the opportunity to write this thesis. I especially want to thank Thomas for endless hours of valuable discussions, without which I would not have been able to write this thesis in this scope, and Richard for his advice on the usage of KeY and his enormous help with the evaluation.

Furthermore, I thank Sebastian Krieter for his support with Fuji and discussions about this thesis.

Finally, I thank my family for all their support.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

In the last couple of years software product lines emerged as a new paradigm in software engineering. Both in the industry and in research, they become more and more popular [Thüm et al., 2014a]. They are used to develop software products that share a common code base, but differ nonetheless gradually. Common and separate code parts are modelled as features (i.e., properties that are visible for end users). The achieved variability leads to increasing complexity regarding the possible variants. They additionally provide cost-efficient reuse of code and the means to automatic product generation [Apel et al., 2013a]. By that, they enormously help dealing with software systems that are getting even more complex. Following that direction, SPL technology is also increasingly used in safety-critical software. In this context program errors are even less tolerable.

For traditionally developed single software products, different means of analysis were established over the years. Formal specification and verification proved as useful instruments to ensure code quality [Hatcliff et al., 2012]. Hence, implementations for these methods are fairly advanced and tools executing them have been established. Specification can be used to define properties of code parts. One specification technique is design by contract [Jézéquel and Meyer, 1997]. It is used in object-oriented and imperative programming to define properties of single modules and by that their behaviour. These definitions are called contracts and for methods they serve as an agreement between caller and callee about how a method functions [Hatcliff et al., 2012]. They at least consist of pre- and postconditions, but can be extended by other elements such as assignable clauses. Preconditions are used to define which properties need to be fulfilled before the respective method can be executed in the expected way. Postconditions define the properties a method guarantees to establish if the caller establishes the precondition. Assignable clauses serve as a list of fields a method can change and can so be used to specify side effects of methods. Contracts only define the behaviour. It is possible to verify the method against its specification with a theorem prover such as KeY [Beckert et al., 2007]. For theorem proving a program is symbolically executed to transform it into a logical formula. This formula is then proven to show the correctness of the program with respect to its contracts [Beckert et al., 2007].

For software product lines, however, analyses are more complex due to their inherent variability and complexity. Different mechanisms have nonetheless been applied to software product lines [Meinicke et al., 2014; Thüm et al., 2014a] for example type checking [Kolesnikov et al., 2013], model checking [Apel et al., 2013e], theorem proving, and combinations thereof [Thüm et al., 2014c]. In order to solve the problems of software product lines, several strategies have evolved. One kind of strategy simply takes the idea of verifying single products and employs it on software product lines. In this product-based strategy, all products of a software product line are generated and proven independently of each other. Using this approach, it is possible to perform verifications with regular theorem provers. Due to the enormous amount of variants, these approaches are infeasible even in small software product lines [Thüm et al., 2014a]. There is also a family-based verification approach in that a metaproduct is generated. This metaproduct encodes the variability of the software product line in the source code and by that transforms the variability at compile time in variability at runtime. However, every time a feature or contract is changed the verification needs to be performed, again.[Thüm et al., 2014a] One last approach is feature-based, meaning that features are verified in isolation. This procedure should prevent a complete re-verification as the code evolves because only the features that changed need to be proven again. However, this strategy has not yet been realized because, as due its notion of isolation, it is impossible to realize interactions between features. Interactions include a method that is defined in one feature is called by a method from another feature [Thüm et al., 2014a].

Some of these approaches have also been combined to overcome weaknesses and increase the benefits [Thüm et al., 2014a]. Both feature-product-based and a feature-family-based approaches were developed. However there is no implementation of feature-family-based strategy yet.

1.1 Goal of the Thesis

The goal of this thesis is to develop and implement a feature-family-based theorem proving strategy. This strategy is aimed to increase proof reuse and hence works in one feature-based and one family-based phase. In the first phase, we analyze features in an isolated manner and therefore transform feature modules into valid Java programs called feature stubs. To be able to verify features with feature module dependencies, we employ the concept of abstract contracts. We then verify the created feature stubs and save the resulting (partial) proofs. In the second phase, we generate a metaproduct. Finally, we verify this metaproduct by reusing the saved partial proofs from phase one and subsequently closing the remaining open proof goals.

We provide tool support for the feature-based verification phase including the generation of the feature stubs. This tool is based on the language-independent composer FeatureHouse [Apel et al., 2013b] that is integrated in FeatureIDE [Thüm et al., 2014b], a development tool for feature-oriented product lines. We further establish a cooperation between FeatureIDE and KeY [Beckert et al., 2007] to provide a fluent verification process. While we use KeY for the verification in both the first and the second phase, our extension to FeatureIDE is used to prepare the feature modules.

Subsequently, we evaluate concept and implementation. We verify a case study with the theorem prover KeY by applying several strategies including a family-based and our feature-family-based approach. Finally, we compare the results with regards to several suitable metrics and discuss found advantages and disadvantages.

1.2 Structure of the Thesis

In Chapter 2, we give an overview about the background the thesis and its contribution. The escribes our approach from a conceptual perspective. Chapter 4 provides a detailed explanation of the implementation of the concept described in Chapter 3 and also includes its limits. The implementation is evaluated by verifying a software product line with different mechanisms and compare relevant metrics in Chapter 5. In Chapter 6, research related to this thesis is examined and presented. Finally, in Chapter 7, we sum up our findings, provide concluding remarks, and questions still open for future research.

2. Background

This chapter introduces the reader to the basic concepts and terminology that are required over the course of this thesis. Section 2.1 gives an overview about software product lines and how they are modelled and implemented. Section 2.2 briefly describes the concept of design by contract and its realization in JML. The chapter concludes with an introduction to theorem proving strategies for software product lines in Section 2.3.

2.1 Software Product Lines

As mass-customization and product individualization arose, so did the demand for ways of highly efficient production of these items. [Apel et al., 2013a; Clements and Northrop, 2001] In software development, the concept of software product lines(SPL) was introduced to match the requirements that emerged in fields such as databases and embedded systems [Beuche, 2003]. They are defined as follows:

> 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. [Clements and Northrop, 2001]

Their main purpose is to allow cost-efficient development of a wide spectrum of products by reusing as much source code as possible [Kang et al., 2002]. In order to achieve these goals a common methodology was established. The product-line engineering process is distinguished into domain engineering and application engineering [Pohl et al., 2005]. Domain, in this context, is defined as follows:

> an area of knowledge that is scoped to maximize the satisfaction of the requirements of its stakeholders, includes a set of concepts and terminology

> understood by practitioners in that area, and includes the knowledge of how to build software systems (or parts of software systems) in that area [Apel et al., 2013a].

However, domain engineering starts with a comprehensive domain analysis which bases upon the knowledge about the specific domain and results in a structured collection of the features covered in the project, usually called feature model. The goal is to determine dependencies between all involved features [Kang et al., 2002]. Depending on the product derivation technique, a feature can represent different aspects of a product line's variability [Apel et al., 2013a]. Due to the nature of this thesis, we focus on the composition-based techniques. In this approach a feature is the module, in which the project's code is to be encapsulated. These modules are called features modules. In the literature, features are defined as follows:

> a characteristic or end-user-visible behaviour of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle [Apel et al., 2013a].

In the domain implementation phase, the necessary artefacts for every feature are created. These artefacts serve as an intermediary result and mainly consist of executables, but can also cover documentation or configuration. In application engineering, the customer's needs and features from the feature model created during domain analysis are used for a requirements analysis to select the features, which are relevant for the final product. Finally, the product is manufactured from implementation artefacts based on the feature selection. The manufacturing can be done manually by the developer. However in most cases products are generated automatically due to the rapidly increasing complexity [Apel et al., 2013a]. In the software product line context, product is defined as follows:

> specified by a valid feature selection (a subset of the features of the product line). A feature selection is valid if and only if it fulfils all feature dependencies [Apel et al., 2013a].

2.1.1 Feature Modelling

There are several ways of representing feature models (e.g., feature diagram, propositional formula), which can be converted into each other [Batory, 2005]. Feature diagrams are trees of features where dependencies can be described through the hierarchy and different kinds of nodes, as well as additional propositional formulas. They are most suitable for humans due their structure. Features with the same parent have a common group type. If their group type is And, regularly all features need to be selected, but more variability can be introduced by giving the possibility to make features either mandatory or optional. The group type OR indicates that at least one of the features needs to be selected. Finally, the group type Alternative only allows for exactly one feature to be selected [Bontemps et al., 2004].

Figure 2.1 shows the feature diagram for the software product line *BankAccount*. Its root feature *BankAccount* as the only mandatory feature serves as a base for all possible products while all other features are optional. If a feature is selected, all its parent features have to be selected as well. If *Transaction* gets selected, the feature *Lock* is automatically selected, too. *TransactionLog* is a feature that implements code relevant for both *Transaction* and *Logging*. Therefore, a custom constraint states that it is selected if *Logging* and *Transaction* are selected.

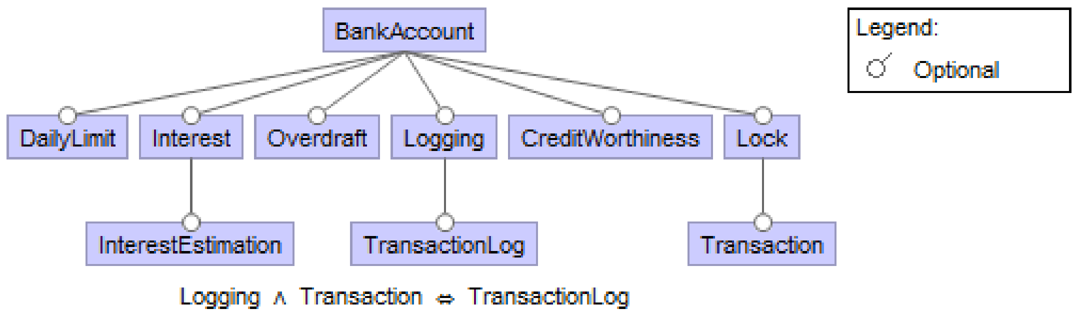


Figure 2.1: Feature Model of BankAccount SPL

However, the feature model can also be modelled by means of propositional formulas, avoiding problems of constraints additional to the actual model, but becoming relatively complex even for yet simple models. Each feature is represented by a variable, which is true if the feature is selected. The overall formula is true if a given selection is valid [Batory, 2005]. The propositional formula in conjunctive normal form for the feature diagram in Figure 2.1 is:

```
        BankAccount
∧ (¬ DailyLimit ∨  BankAccount)
∧ (¬ Interest ∨ BankAccount)
∧ (¬ Overdraft ∨ BankAccount)
∧ (¬ Logging  BankAccount)
∧ (¬ CreditWorthiness ∨ BankAccount)
∧ (¬ Lock ∨ BankAccount)
∧ (¬ InterestEstimation ∨ Interest)
∧ (¬ TransactionLog ∨ Logging)
∧ (¬ Transaction ∨ Lock)
∧ (¬ Logging ∨ ¬ Transaction ∨ TransactionLog)
∧ (¬ TransactionLog ∨ Logging)
∧ (¬ TransactionLog ∨ Transaction)
```

Listing 2.1: Propositional Formula for BankAccount SPL

2.1.2 Product Generation

A subset of all features of the domain, called configuration, is used to generate products. Only if the configuration is valid, a product will be created. The process of product generation can be mathematically described as a series of applications of the operator • on a set of features [Apel and Lengauer, 2008].

$$p = f1 \bullet f2 \bullet ... \bullet fn \tag{2.1}$$

The operator • represents a function over features used to compose them. It is defined as follows: [Apel and Lengauer, 2008]

$$\bullet : F \times F \rightarrow F \tag{2.2}$$

In generative programming [Czarnecki and Eisenecker, 2000], several techniques such as aspect- [Kiczales et al., 1997; Mezini and Ostermann, 2004; Wampfler, 2007], delta- [Haber et al., 2012; Schaefer et al., 2010], or feature-oriented [Apel and Kästner, 2009; Mezini and Ostermann, 2004] programming employ this approach of feature composition. As this thesis' focus is feature-oriented programming, it is further introduced in Section 2.1.3.

2.1.3 Feature-oriented Programming

Feature-oriented programming (FOP)[Prehofer, 1997] is a composition-based programming paradigm for implementing software product lines. It is mostly used for object-oriented programs, but its mechanisms can also be applied to functional programming. In FOP, features are implemented by the means of feature modules. Each feature module contains all artefacts relevant to a feature and hence represents a one-to-one mapping of a feature [Apel et al., 2013a].

Superimposition [Apel and Lengauer, 2008] is a concept employed to compose feature modules. Domain artefacts are merged along the structures of their underlying paradigm. So, in object-oriented programming, classes and methods are merged with methods or classes in other features that have the same name [Apel and Lengauer, 2008]. The order in which they are merged is usually assumed to be a total order and needs to be defined beforehand. The process can also be referred to as refinement, because when two features are merged, the second one refines the classes, methods, fields of the first one [Apel et al., 2013a].

The idea of superimposition is implemented by FeatureHouse [Apel et al., 2013b], an open-source framework and tool chain for feature-module composition by employing language-independent feature structure trees (FST). Using the software product line seen in Figure 2.1, we show how the feature composition with superimposition works. Listing 2.2 is part of the software product line shown in Figure 2.1 and is a role in the feature *Bankaccount*. It represents the core implementation of the account and serves as the introduction for class `Account` in the software product line. The feature *DailyLimit* implements a maximum amount of money that can be transferred per day.

We show its implementation of role `Account` in extracts in Listing 2.3. The role introduces two new fields `withdraw` and `balance` and refines the methods `update` and `undoUpdate` (not included in listing).

```
public class Account {
    public final int OVERDRAFT_LIMIT = 0;
    /*@ public invariant this.balance >= OVERDRAFT_LIMIT; */
    public int balance = 0;

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

    /*@ requires x != 0;
     @ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x);
     @ assignable balance; @*/
    boolean update(int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT\_LIMIT)
            return false;
        balance = balance + x;
        return true;
    }

    /*@ ensures (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) − x);
     @ assignable balance; @*/
    boolean undoUpdate(int x) {
        int newBalance = balance − x;
        if (newBalance < OVERDRAFT\_LIMIT)
            return false;
        balance = newBalance;
        return true;
    }
}
```

Listing 2.2: Role *Account* of Feature *BankAccount*

```
class Account {
    public final static int DAILY_LIMIT = −1000;
    /*@ public invariant withdraw >= DAILY_LIMIT; */
    public int withdraw = 0;

    /*@ requires \original; 17
     @ ensures \original;
     @ ensures (!\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;
16          }
17          if (!original(x))
18              return false;
19          withdraw = newWithdraw;
20          return true;
21      }
22      [..]
23  }
```

Listing 2.3: Role *Account* of the Feature *DailyLimit*

The result of the composition of the roles shown in Listing 2.2 and Listing 2.3 can be seen in Listing 2.4. The two implementations of the method `update` are separated into two methods. The last refinement of a method keeps its name while the others are extended to indicate their respective feature. It is possible to call the originating implementation from within a refinement by use of the keyword **`original`**. As can be seen in Line 17 of Listing 2.3 the method `update` in DailyLimit uses the keyword. Therefore the keyword is replaced by a call of the *BankAccount* implementation of the method during composition.

```

public class Account {
    public final int OVERDRAFT_LIMIT = 0;
    /*@ public invariant this.balance >= OVERDRAFT_LIMIT; @*/
    public int balance = 0;
    /*@ ensures balance == 0; @*/
    Account() {}

    /*@ requires x != 0;
    ensures (!\result ==> balance == \old(balance))
      && (\result ==> balance == \old(balance) + x); @*/
     private boolean  update__wrappee__BankAccount  (int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT) return false;
        balance = balance + x;
        return true;
    }

    /*@ requires ( x != 0 );
    ensures ( (!\result ==> balance == \old(balance))
      && (\result ==> balance == \old(balance) + x) );
    ensures (!\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 (!update__wrappee__BankAccount(x)) return false;
        withdraw = newWithdraw;
        return true;
    }
    [...]
}
```

Listing 2.4: Composition of Features *BankAccount* and *DailyLimit*

However, constructors are composed differently. There can be only one constructor with the same set of parameters for each class. Hence, all refinements of a constructor are merged into one constructor in the product. In the resulting constructor, the several lines of the refinements are ordered as the features the refinements originally belonged to.

Methods can be refined and so different features can add to a behaviour of method. For fields, most refinements are ignored, when the fields are composed. When several features define a field, only the last feature's implementation is used in the derived product.

2.2 Feature-oriented Specification of Software Product Lines

Specification serves to define properties of a the system and therefore improves the quality of the code [Hatcliff et al., 2012]. Languages and formalisms vary widely (e.g., JML [Burdy et al., 2005] for Java, SPEC#[Barnett et al., 2011] for C#) . Here, due to its importance for this thesis, design by contract and some of its implementations are further explained.

Design by contract uses assertions which provide predicates that a program needs to fulfil [Jézéquel and Meyer, 1997; Meyer, 1992]. The different kinds of assertions, namely class invariants and method contracts, extend the respective module used in object-oriented programming they accompany. Class invariants are required to hold at any publicly visible state of a class [Hatcliff et al., 2012]. Method contracts work as commitments between caller and callee and typically consist of pre-, postconditions and assignable clauses [Hatcliff et al., 2012; Meyer, 1992]. Preconditions represent predicates which hold before a method's execution, while postconditions must hold after the execution and are to be assured by the called method itself. Assignable clauses are lists of fields that can be changed by the accompanying method.

The Hoare notation [Hoare, C. A. R., 1969] serves as formal system in which contracts can be represented, but is not tied to a specific programming language. Method contracts are realized as follows [Hoare, C. A. R., 1969]

```
C = {P}Q{R}
(C := contract; P := precondition; Q := program; R := postcondition)
```

The expression can be interpreted as that if P was true before the program Q started, R will be true after its execution. On the other hand if P is false, the state of R is irrelevant, as R is only guaranteed, if P holds.

The Java Modeling Language (JML) [Burdy et al., 2005; Leavens et al., 2006] implements the design by contract concept for Java [Leavens and Cheon, 2006]. Assertions are defined in Java comments with @-symbols at beginning of every line. Specific keywords indicate what kind of assertion a JML statement is representing.

In Listing 2.2, we show an exemplary Java class with specifications in JML. The class invariant in Line 3 states that the amount of money in the account needs to be greater than the `overdraft` the account is allowed to have. Method `update` is accompanied by a contract starting in Line 9. Its precondition indicated by the keyword **`requires`** demands that the parameter x is not zero. In the following two lines starting with the keyword **`ensures`**, the postcondition is defined. It states that if the transaction was successful the balance of the account needs to be different after the method execution, but if the transaction was not successful balance must not have been changed. Line 12 defines the assignable clause and states that the method is allowed to only change the field `balance`.

To realize specification in software product lines it is not only necessary to be able to specify the behaviour of features but also to transform them into the generated product. Therefore, it is necessary to compose them as well. Several approaches emerged to accomplish this composition including cumulative, conjunctive, consecutive and explicit contract refinement [Thüm et al., 2012b]. We only give a short overview of explicit contract refinement because it is the most relevant to this thesis.

In explicit contract refinement, a refining contract simply replaces any contract defined beforehand [Thüm et al., 2012b]. It is however possible to completely or partially include a former specification by using the keyword **`original`**. Listing 2.3 provides an example both for explicit contract refinement and for the use of **`original`**. The composed contract of method `update` then also consists of the pre- and postconditions of the originating methods' contracts. Thus, it works similar to method refinement in feature-oriented programming.

2.3 Verification of Software Product Lines

Generally, verification processes can vary regarding subjective of verification and used algorithms. Possible verification techniques include model checking, type checking, static analysis, and theorem proving [Ehrenberger, 2002]. In this thesis, we focus on theorem proving and, for an easier wording, use the terms verification and theorem proving interchangeably. In the following, theorem proving is described further.

Using specifications as realized by formal languages such as JML, theorem proving verifies the correctness of a program with respect to its contract. The verification is performed by theorem provers such as KeY [Beckert et al., 2007] and can be achieved automatically and/or by user interaction. One possible way theorem provers perform a verification is to symbolically execute the program in order to transform it into first-order propositional formulas, called proof goals, that can be verified.

During symbolic execution, theorem provers can work in two different ways, when it comes to method contracts of called methods [Beckert et al., 2007]. First, the body of the called method can be inlined. In this case, the called method's contracts are ignored and its actual implementation is included into the verification. This technique, however, can produce large proofs as all called methods become part of the proof [Beckert et al., 2007]. The second way is to use the contract of the called methods. This way, methods only need to be verified once, because they are not potentially symbolically executed several times when they get inlined into other proofs [Beckert et al., 2007]. Additionally, if a called method's source code is not available (e.g., when using external libraries) its contract can be used instead [Beckert et al., 2007]. In contrast to the first technique however, a method's verification is only completed when the verification of all methods it calls is completed, because the called method's contract may not accurately describe the method's behaviour.

Three main approaches were developed to deal with theorem proving in software product lines: product-based, family-based and feature-based [Thüm et al., 2014a]. In the product-based approach, all products are built and verified individually [Thüm et al., 2014a], which can cause massively redundant verification as different products can consist of similar feature combinations. On the other hand the approach is easy to realize and already existing theorem provers can be used [Thüm et al., 2014a].

In family-based verification, the implementation of the feature modules are translated into one metaproduct. This metaprocut simulates the product line's behaviour.[Thüm et al., 2014a] Furthermore, the specification of the feature modules is transformed into a metaspecification encompassing the whole software product line. This process transforms the variability at compile time into variability at runtime. Thanks to this variability encoding it is possible to prove the software product line by solely proving the metaproduct instead of all single products [Thüm et al., 2014a, 2012a]. Compared to product-based approaches they have several advantages. So it is not necessary to generate all products which avoids hugely redundant verifications in more complex product lines. On the other hand it is necessary to re-verify the software product line as the code [Thüm et al., 2014a]. Furthermore family-based strategies can require large amounts of memory for software product lines.

Feature-based approaches rely solely on *features* and ignore the variability. The code base of a single feature is verified in isolation (i.e., potential feature interactions are ignored during the analysis process). The upsides are that analyses can be done modularly and therefore do not need to consider the behaviour of other features. Hence, less memory is consumed than in family-based approaches and as code evolves only changed features need to be re-proven. However, as feature interactions occur frequently in software product lines [Apel et al., 2013c], a solely feature-based approach for theorem proving is not possible [Thüm et al., 2014a].

In order to overcome disadvantages, strategies have been combined [Thüm et al., 2014a]. Due to the goal of this thesis, we focus on feature-family-based approaches below. Feature-family-based approaches aim to reuse proofs for features for the verification of the whole product line. Both redundant verification and restriction to product lines without any interactions between features are so to be avoided. [Bubel et al., 2014; Hähnle and Schaefer, 2012; Hähnle et al., 2013a]

3. Feature-Familiy-Based Theorem Proving of Product Lines

In the previous chapter, we introduced several strategies to verify software product lines and presented different advantages and disadvantages over each other. In this chapter, we present a new approach combining already existing strategies to both exploit existing benefits and overcome their weaknesses. In particular, we combine a feature-based and a family-based approach of theorem proving and develop a feature-family-based verification strategy. With this strategy, we aim to fully verify a software product line's behaviour specified by JML method contracts. Simultaneously, we want to increase the reuse potential of *partial proofs* during code evolution. This increase is to be achieved through the feature-based phase because only the features that were not proven yet or whose proof is not up-to-date need to be proven again. The feature-based phase can also work as a fallback when the family-based theorem proving becomes to consumptive of resources that it cannot be performed completely.

Thüm et al. [2012a] showed, that it is possible to encode a software product line's variability in the source code and, by that, verify it with regular theorem provers suited for single products. We aim to do similar in both the feature-based and the family-based phase of the verification. Considering that, we develop a technique to create valid Java programs from feature modules in the first phase. Valid here means that it is compilable by the Java compiler and can be loaded into a theorem prover without error. In the second phase, we create a metaproduct incorporating a product line's variability. Therefore, we adapt the mechanism developed by Thüm et al. [2012a] and Meinicke [2013] to fit our needs. Our approach thereby is focussed on explicit contract refinement [Thüm et al., 2012b] because of its similarities feature-oriented to method refinement.

In Section 3.1, we describe our feature-based approach to theorem proving of software product lines. We explain, how the results of this first phase can be used in Section 3.2, where we describe the family-based part of our approach. In Section 3.3, we sum up our concept.

3.1 Feature-Based Theorem Verification

Our goal in this phase is to prove the behaviour of feature modules with regular theorem provers. We therefore present our approach of creating regular programs from feature modules in this section. Additionally, we describe the theorem proving process itself and discuss aspects worth to be considered regarding this process.

The feature modules serve as input for our algorithm. As stated before, the algorithm should result in a valid Java program. Furthermore, we only consider features in an isolated manner in this phase. This isolation is rather difficult because features regularly interact with each other. Feature interactions can be realized on purpose (e.g. when a method calls a method from a different feature), but may also occur unintentionally or even against the intent of the developer [Calder et al., 2003]. A lot of research is performed to detect unwanted feature interactions [Apel et al., 2013d, 2010b; Calder and Miller, 2006; Scholz et al., 2011]. To better distinguish undetected and unwanted feature interactions from the explicit method calls, we name the latter *feature module dependencies*. Our algorithm's result should so also include the possibility to handle these explicit dependencies. We realize that by relying on the concept of feature stubs, as introduced by Kolesnikov et al. [2013]. A (feature) stub is:

> a bundle of Java interfaces and classes, possibly with member prototypes, that represent the types and members a feature requires from other features.

Feature stubs contain additions to feature modules so that they become valid Java programs. Kolesnikov et al. [2013] employed them as a basis for feature-based type checking and we extend the concept such that it is suitable for theorem proving. We not only need to provide additions to implementation but also to specification, to keep the behaviour definitions with contracts semantically equivalent. Section 3.1.1 and Section 3.1.2 explain the generation of feature stubs in more detail.

3.1.1 Generation of Feature Stubs for Feature Modules

The algorithm takes feature modules as input and creates temporary folders for each of them. To localize the access of elements that are not defined in the feature, We then perform a type check for all roles of all feature modules in the temporary folder. These checks include type access, method access, keyword original, field access, access to elements only accessed in JML statements, and the usage of external libraries. Furthermore, we provide some additions to make contracts more precise and simplify the verification

We now explain these cases in more detail and use the Java-based BankAccount SPL introduced in the last chapter for illustration. We further employ different source code extracts from the SPL that demonstrate the respective aspect. Additionally to the features *BankAccount* and *DailyLimit*, which were already introduced in Chapter 2, we now examine the features *CreditWorthiness* and *Transaction*.

Feature *CreditWorthiness*, which is part of the BankAccount SPL, introduces a new method `credit` *Account* of the feature. Method `credit` checks whether the balance of an account gets smaller than zero, if a specific amount is subtracted.

Transaction, a feature in the software product line, transforms money transfers from different accounts into an atomic operation. A new role Transaction is introduced, which implements the methods `transfer` and lock to realize this functionality by relying on the locking functionality of feature *Lock*.

We show many examples to illustrate the generation, structure and elements of the feature stubs. Nonetheless, we can only show the parts of the feature stubs that are relevant to the respective point we discuss in this part. If we showed feature stubs completely, the overall readability would be impaired. As we want do fully document our procedure and results, we show listings of the full feature stubs of features *BankAccount* and *TransactionLog* in Appendix A and provide additional explanations.

Access to Types

With theorem proving, we can verify a program's behaviour. However, a non-compilable program's behaviour is undefined. Therefore, the feature stubs serving as input for the theorem prover need to be type-safe. We establish the type-safety of the feature stubs with a type check of the software product line. If a software product line does not contain any type errors, it can provide all types, methods and fields to realize a type-safe feature stub. If, however, a product line is not type-safe, we do not need to continue the verification, as its behaviour is undefined. A software product line is type-safe, if all its possible variants are type-safe.

As with theorem proving, several strategies have emerged to perform type checking for software product lines such as product-based [Apel et al., 2008; Istoan, 2013], family-based [Apel et al., 2010a; Kästner et al., 2012; Kolesnikov et al., 2013], feature-product-based and feature-family-based approaches. Feature-based approaches do, again, not provide enough information for a type check of a software product line [Thüm et al., 2014a], because they only consider features in isolation and do not recognize any interactions between features. Product-based and family-based approaches can provide the necessary information. However, product-based approaches require the generation of all products, which leads to redundant analysis and aggregation of data over the generated products [Thüm et al., 2014a]. Family-based type checking forces us to include information that go beyond our respective feature, but they are necessary to even perform the theorem proving. Furthermore, compared to theorem proving, a type check takes only an insignificantly small amount of time and resources.

We employ the results of a sufficient type check to establish type safety for each feature stub. If a type is not part of the feature, but part of the software product line, a class prototype for that type is created in the respective feature stub.

Example 3.1.
In Listing 3.1, we show the role Transaction of feature Transaction in extracts. It contains method `transfer`*, which has two parameters of the type* `Account` *(see Line 13). Feature Transaction does not have a role Account originally. In accordance to our approach, an empty class prototype* `Account` *is created to provide a match for that type access.*

```
 1 public class Transaction {
 2     /*@ requires destination != null && source != null;
 3       @ requires source != destination;
 4       @ ensures \result ==> (\old(destination.balance) + amount ==
 5         destination.balance);
 6       @ ensures \result ==> (\old(source.balance) - amount ==
 7         source.balance);
 8       @ ensures !\result ==> (\old(destination.balance) ==
 9         destination.balance);
10       @ ensures !\result ==> (\old(source.balance) == source.balance);
11       @ assignable \everything;
12       @*/
13     public boolean transfer(Account source, Account destination, int
14     amount) {
15         if (!lock(source, destination)) return false;
16         try {
17             if (amount <= 0) {
18                 return false;
19             }
20             if (!source.update(amount * -1)) {
21                 return false;
22             }
23             if (!destination.update(amount)) {
24                 source.undoUpdate(amount * -1);
25                 return false;
26             }
27             return true;
28         } finally {
29             source.unLock();
30             destination.unLock();
31         }
32     }
33     [...]
34 }
```

Listing 3.1: Role *Transaction* of Feature *Transaction*

Access to Methods

A contract defines a method's behaviour. With theorem proving, we can verify if a method behaves according to its contract. If a method is called in a feature, but not defined within this feature, a prototype for that method is created in the role the method would be defined in. This is the case, when a method is introduced in one feature, but not refined in another feature, in which it is called. We also add a comment to the method prototype to indicate that it was added for the feature stub.

There are different approaches to a method prototype's contains. If the signature of the prototype indicates a return type other than **`void`**, the prototype needs to include a return statement. For primitive types, this statement is a value that is part of the type (e.g. `0` for type **`int`**). For reference types, on the other hand, the method prototype can either return **`null`** or a newly created object of said type. The return statement of the method prototype may, however, be included in a proof of the method calling the method prototype. Therefore, it may be more desirable to not include a return statement, at all. If the theorem prover, which is used for verification, accepts incomplete programs as input, the return statement may be omitted.

Example 3.2.
Method `transfer` in Listing 3.1 calls method `update`, which belongs to role Account, but is not defined in feature Transaction. Therefore, a method prototype `update` is created in the role Account for the feature stub. In Listing 3.2, we show the resulting class in part. In this example, we choose to exclude return statements. The listing also shows the contract that is created for the method prototypes. We refer to the listing again in Section 3.1.2 and explain the generation of the contract for method prototypes.

```
public class Account{
    /*field prototype*/
    public int balance;
    [...]

    /*method prototype*/
    /*@ requires_abs updateR;
    @ ensures_abs updateE;
    @ assignable_abs updateA;
    @*/
    boolean update(int x) { }
    [...]
}
```

Listing 3.2: Role *Account* of Feature Stub for Feature *Transaction*

Access to Method with Keyword Original

Due to superimposition in feature-oriented programming, methods and fields can also be refined. Methods can call previous implementations of themselves during their own execution by means of the keyword **`original`**. In that case, a method prototype is created and the keyword is then replaced by a call of that method. The method prototype's name includes the original method's name, the keyword original and the current feature, each separated by an underscore to indicate their origin.

Example 3.3.
The Listing 2.3 on Page 9, which we used in the last chapter to explain this very mechanism of calling previous implementations of a method, illustrates this case. Method

*`update` uses the keyword **`original`** to call a previous implementation of itself. Therefore a method prototype with the name `update_original_DailyLimit` is created in the feature stub and the call of **`original`** is replaced by a call to that method. In Listing 3.3, we show both the created method prototype (see Line 21) and the replaced call (see Line 10).*

```
 1 class Account {
 2     [...]
 3     boolean update(int x) {
 4         int newWithdraw = withdraw;
 5         if (x < 0)  {
 6             newWithdraw += x;
 7             if (newWithdraw < DAILY_LIMIT)
 8                 return false;
 9         }
10         if (!update_original_DailyLimit(x))
11             return false;
12         withdraw = newWithdraw;
13         return true;
14     }
15     [...]
16 
17     /*method prototype*/
18     /*@ requires_abs   update_original_DailyLimitR;
19     @ ensures_abs    update_original_DailyLimitE;
20     @ assignable_abs update_original_DailyLimitA;@*/
21  boolean update_original_DailyLimit(int x) { }
22 }
```

Listing 3.3: Role *Account* of Feature Stub for Feature *DailyLimit*

Access to Fields

Fields are treated similarly to methods. If a field is used in a feature's role, but not defined within the feature, a prototype is created to represent it. However, fields that do not even belong to the software product line need to be treated differently. We postpone the discussion of this issue to the paragraph about access to external libraries. The field prototype is created in the role, the field would actually belong to, and is accompanied by a comment that indicates its purpose.

```
class Account {
    /*@ requires amount >= 0;
     @ ensures balance >= amount <==> \result;
     @ assignable \nothing;
     @*/
    boolean credit(int amount) {
        return balance >= amount;
    }
}
```

Listing 3.4: Role *Account* of Feature *CreditWorthiness*

Example 3.4.
In Listing 3.4, we provide an example for the creation of field prototypes. Method `credit` uses field `balance` although it is not defined in feature CreditWorthiness. For the feature stub of Creditworthiness, a field prototype`balance` is generated. The result can be seen in Listing 3.5.

```
class Account {
   /*@ requires_abs credit_CreditWorthinessR;
     @ def credit_CreditWorthinessR = amount >= 0;
     @ ensures_abs credit_CreditWorthinessE;
     @ def credit_CreditWorthinessE = balance >= amount <==> \result &&
     @ FM.FeatureModel.CreditWorthiness;
     @ assignable_abs credit_CreditWorthinessA;
     @ def credit_CreditWorthinessA = \nothing;
     @*/
    boolean credit(int amount) {
        return balance >= amount;
    }
    [...]
    /*field prototype*/
    public int balance;
}
```

Listing 3.5: Role *Account* of Feature Stub for Feature *CreditWorthiness*

Access to a Type, Method and Field Only from within JML Statements

The cases described above deal with types, methods and fields that are accessed in the implementation of features, they are not defined in. It is, however, possible, that fields, methods, and types are, while defined in one feature's implementation, accessed in another feature's specification. For these elements, we also need to generate prototypes - depending on which element is accessed a class, field, or method prototype.

Example 3.5.
For illustration, we show role `Transaction` of feature Transaction in Listing 3.1 as an example for a field that is accessed in a different feature's specification. The contract of `transfer` accesses field `balance` (see Line 5 and Line 7) that originally belongs to role `Account`, but is not defined in feature Transaction. In accordance with our approach, we create a field prototype `balance` in role `Account`. We show the respective part of the feature stub in Listing 3.2.

Access to External Libraries

The definition of feature stubs explicitly states that the extensions a feature stub contains are only from within its produt line. If a method, field, or class from an external library is used, a feature stub is not able to provide the necessary extensions. Using external libraries becomes a problem, as the respective feature cannot be transformed into a feature stub. We propose two possible approaches. Either the verification is aborted because it is not possible to conclusively prove the stub's functionality, or the source code including contracts has to be provided. In the latter case, the verification process can be expanded to the external libraries and, thereby, completed. The expansion can, depending on the support by the theorem prover, include the external library in the verification or assume all used elements from the library to be verified.

Generation of FeatureModel Class

We also propose to create a class `FeatureModel` containing a boolean field representing the original feature of the feature stub. We use the field for a more precise specification to simplify proofs. The class may, however, be omitted, if the precision is not needed. We discuss the specification in more detail in Section 3.1.2.

Example 3.6.
We show the `FeatureModel` *class for feature DailyLimit in Listing 3.6. The class only contains the boolean field* `DailyLimit`*, that represents the feature, the feature stub is created for.*

```java
package FM;
public class FeatureModel {
    public static boolean DailyLimit;
}
```

Listing 3.6: Class *FeatureModel* For Feature Stub of Feature *DailyLimit*

3.1.2 Generation of Feature Stubs for Feature-oriented Contracts

Above, we pointed out, that methods can access methods, fields and types, which are not defined in the method's feature. To realize such *feature module dependencies*, we generate method prototypes. For theorem proving, we not only need other features' methods but also their contracts. However, in the feature-based verification phase, we cannot access the contracts of methods, which are not defined in the feature that is to be verified. Therefore, we employ the concept of abstract contracts. This concept was developed by Bubel et al. [2014]; Pelevina [2014]. The goal of abstract contracts is to increase potential of reusing proof parts of regular object-oriented programs. Instead, we use abstract method contracts to realize *feature module dependencies* for feature stubs.

Abstract method contracts are defined as [Pelevina, 2014]:

```
@ requires_abs placeholdernameR;
@ ensures_abs placeholdernameE;
@ assignable_abs placeholdernameA;
@ def methodR = <JML expression> ;
@ def methodE = <JML expression>;
@ def methodA = <List of fields that can be changed by method>;
```

The first three lines are the abstract section of the abstract contract. The keywords **`requires_abs`**, **`ensures_abs`**, and **`assignable_abs`** represent the abstract definitions of the requires, ensures, and assignable clauses including the names of the placeholders. The actual content of the concrete method contract is provided by the last three lines. They are called concrete section and contain the concrete definitions of the placeholders. We generate the placeholders' names by concatenating the method's name, an underscore, the feature's name, and a capital `R` for requires, a capital `E` for ensures or a capital `A` for assignable.

We aim to increase the reuse potential of verification results. Therefore, we transform all contracts of methods, originally defined in the feature, into abstract contracts, so that the proofs created are with respect to the placeholders defined in the abstract contracts. We use the concrete section of the contracts for the original clauses of the contracts. For contracts that consist of more than one **`ensures`** or **`requires`** clause we compose these clauses by means of logical Ands (`&&`) into one clause. For method contracts, which lack a clause, we only generate the declaration of the placeholder for that clause.

Example 3.7.
For illustration, we give an example of the transformation of a concrete contract in the feature module into an abstract contract for the feature stub. In Listing 3.4, we show role Account of feature `CreditWorthiness`. The role contains method `credit` and its contract. In Listing 3.5, we show the feature stub, generated for this role. Method `credit`'s contract is transformed into an abstract contract. In Lines 2, 4, and 7 of Listing 3.5, we show the abstract section of the abstract contract (i.e., the declaration of the placeholders). Lines 3, 5, and 8 contain the concrete sections (i.e., the definition of the contract.

For methods that originate in the feature of the feature stub, we also add an **`requires`** clause. This **`requires`** clause states that the feature variable, defined in `FeatureModel` Class, has to be true. We propose to add this clause, as it limits the possible frame of a method. This limitation makes the verification of the method easier. However, as with the `FeatureModel` Class, this clause is not necessary and can be omitted.

Example 3.8.
*For illustration of additional **`requires`** clause, we present an example. Therefore, we refer again to Listing 3.4 and Listing 3.5. The contract of method `credit` is enriched by the additional **`ensures`** clause `FM.FeatureModel.CreditWorthiness`, which states, that the feature variable `CreditWorthiness` must be **`true`**. The clause is composed with method `credit`'s original **`requires`** clause by means of a logical And.*

As explained above, when creating the feature stubs, called methods, which are not defined in the current feature, are created in form of a method prototype. Additionally, for each created method prototype a contract is created that only consists of the abstract section. We only need the abstract section because it includes the declaration of the placeholders. During the verification, the placeholders can be integrated into the proof and the proof can be performed with respect to them, so that at least a *partial proof* can be created. When there is a definition of the placeholder, the proof can be completed.

Example 3.9.
In Listing 3.1, we show an example for the generation of abstract contracts for method prototypes. Method `transfer` calls method `update`, which belongs to role Account. As there is no refinement of method `update` in feature Transaction, a prototype of method is created as explained in Section 3.1.1. We enrich the prototype with the abstract section of a contract. The result can be seen in Listing 3.2 in Lines 7 to 9.

When using Explicit Contract Refinement, it is possible to indicate that a specification of a previous implementation should be included by using the keyword **`original`** in a method contract. The keyword can both be used in pre- and in postconditions. The use of the keyword represents a *feature module dependency* as a specification of another feature's method is referenced. There are several ways of dealing with this issue. First, respective methods are ignored during the feature-based verification because not all relevant information contained by the keyword *original* are not provided. To make sure, the methods are skipped during verification the contract can just be omitted for the feature stub generation. This way, a theorem prover does not even recognize this method as to be verified. However, if methods whose contracts contain the keyword **`original`** are skipped, there are not even *partial proofs* that can potentially be reused later. Second, the contract of the method could be transformed into an abstract contract only including the abstract section of the contract. With this abstract section, it would be possible to at least partially prove the method. Nonetheless, we lose information, as the abstract section only declares the placeholder, but not defines them. A third way of dealing with the keyword is by using the placeholders created with method prototypes as representatives. When a method contract uses **`original`**, the keyword is replaced by the placeholder to indicate it depends either on the other method's pre- or postcondition. This approach requires the theorem prover to support using placeholders in other methods' contracts, though.

Example 3.10.
*We show an example for the use of the keyword in Listing 2.3 on Page 9. The contract for the method update uses **`original`** both for its pre- and its postcondition. In accordance with our first discussed approach, the `update`'s contract could just be omitted. In our second approach, we replace the concrete contract of `update` by the abstract section of an abstract contract. We show the result in Listing 3.7.*

```
class Account {
    public final static int DAILY_LIMIT = -1000;
    public int withdraw = 0;

    /*@ requires_abs update_DailyLimitR;
      @ ensures_abs update_DailyLimitE;
      @ assignable_abs update_DailyLimitA;
      @*/
    boolean update(int x) {
        int newWithdraw = withdraw;
        if (x < 0)  {
            newWithdraw += x;
            if (newWithdraw < DAILY_LIMIT)
                return false;
        }
        if (!update_original_DailyLimit(x)
            return false;
        withdraw = newWithdraw;
        return true;
    }
    [..]
}
```

Listing 3.7: Role *Account* of Feature Stub For Feature *DailyLimit*, Part 2 - Abstract Contract

3.1.3 Feature-Based Theorem Proving

Every automatically created feature stub is to be proven with a theorem prover. As we only consider each of the feature stubs in isolation, we cannot cover explicit configurations but only the behaviour within the feature itself. Nevertheless, it is possible to verify base features that do not have any *feature module dependencies* to other feature modules completely. While methods without these *dependencies* may be verified completely, the feature-based theorem proving phase does not result in a conclusive verification all methods in features that have *feature module dependencies*.

With a theorem prover, methods are verified individually. A method's contract is deemed to be proven when a theorem prover can show that if the precondition is held, when the method is executed. Additionally, the method needs to be correctly executed and its postcondition must be satisfied after having finished. Only methods originating in the feature currently to be proven actually need to be proven. That is, method prototypes can be ignored because they do not belong to this feature.

During the verification process of each feature, the abstract sections are to be used as a placeholder for the actual method. Thereby, method calls from other features can be simulated with these contracts.

There are several cases, in which a proof goal cannot be closed. First, it is possible that the contract does not actually describe the method's behaviour (i.e., the method does not fulfils its specification). Second, the method does fulfil its specification, but the theorem prover is not able to prove it. In both cases the proof is not closed completely and the saved proof parts do not provide any benefit. Third, a contract may rely on an abstract contract that does not have a concrete section. It therefore cannot be closed at this point because the contract lacks the concrete definition of the placeholders. In this case, the *partial proof* can be saved for a potential later reuse.

As mentioned above, we only consider features in isolation and, by that, are not able to find an error in a specific configuration. However, we can determine that, if a method without *dependencies* to other feature modules (as described in Section 3.1.1) cannot be verified, either the method's behaviour and its contract do not match or the theorem prover is not powerful enough to prove the contract.

3.1.4 Re-Verification after Code Evolution

When the code changes, so may the behaviour of a program. A change in a software product line may change the behaviour of several of its variants. It may therefore be necessary to re-verify the product line. A re-verification is only necessary when the feature stub of a feature module would actually be different from the feature stub the last proof is based on. In that case the last proof is invalidated. There are, however, changes that do not invalidate a feature stubs's proof.

The issue of change detection in software product lines can be traced back to *change impact analysis* [Passos et al., 2013a,b]. *Change impact analysis* is used to determine the effects of a change in the code. For software product lines, it can help developers to determine whether a change introduces inconsistencies and to decide whether the change should be applied nonetheless [Passos et al., 2013a]. Several aspects, both adopted from regular programs and originating in the field of software product lines, need to be considered regarding this question. Subsequently, we only discuss some examples to illustrate the issue and spotlight a few relevant aspects.

A re-verification is necessary when implementation or the contract of a method changes. The method's proof may get invalidated because either the behaviour of the method or the definition of its behaviour changes. Furthermore, if a method calls a method outside its own feature and the callee's signature is changed, the method's current proof may also be invalidated. In most cases, this change results in an implementation change in the caller's feature as well (e.g. when a parameter is added to the method). However, when only the return type or a parameter's type of the callee is changed, calls do not necessarily change (e.g. when the return type is changed to a superclass).

However, it is not always necessary to re-verify a feature. First, if only the order of methods or fields is changed, the generated feature stub is behaviourally equivalent. Additionally, as theorem provers do verify each method individually, the methods' order in the source code is not relevant for them. Second, when an method is renamed, usually neither the method itself nor its callers do have to be re-verified because both its behaviour and its contract stay the same. However, if a *partial proof* has been saved for a later reuse, either a re-verification of the method and its callers or an update of the *partial proofs* is necessary. Finally, changes regarding the feature model or the composition order do not invalidate a feature stub's proof. They can result in a type error (e.g. if a feature that contains a refining method calling a previous implementation is arranged before the introductory feature). In case of a type error, as we stated above, a verification is useless and we do not start the creation of feature stubs. If no type error occurs, we do not need to re-verify because in feature-based verification other features are ignored.

As mentioned above, we do not provide a full discussion on the topic of change impact analysis, but only give a few examples to illustrate its relevance. The discussion can easily extended to invariants and other JML constructs, or the type hierarchy in the feature stubs.

3.2 Feature-Family-Based Theorem Proving

In this phase, our goal is to verify the product line as a whole. We aim to do this by reusing the *partial proofs* that were the result of the first phase. As we build upon the first phase, we assume the software product line to be already type checked and free of compilation errors. Again, we employ a regular theorem prover for the verification and, include the variability model of the software product line. We build a metaproduct that can be handled by a regular theorem prover. We further adapt the *partial proofs* obtained in the feature-based verification in that manner that they can be (re-)used on the metaproduct.

A metaproduct realizes one possible way of family-based verification [Thüm et al., 2014a]. It is used to simulate all products of a software product line and, to do that, includes all domain artefacts of all features.

3.2.1 Generation of the Metaprogram

Our algorithm is based on the metaproduct generation developed by Thüm et al. [2012a] and implemented and extended by Meinicke [2013] as their requirements are close to ours. They used the metaproduct for a family-based verification. We adapt the generation process as needed for our feature-family-based verification approach. In the following, we explain the parts, we adopt from Meinicke [2013]. In the second part, we describe the differences between the approach and ours in more detail.

Adopted Aspects from Meinicke [2013]

Meinicke [2013] used a class `FeatureModel` introduced by Thüm et al. [2012a] that represents the feature model, which includes a boolean field per feature, also known as feature variables. The class is used for the realization of runtime variability as the fields are used to save the features' state. The feature variables are used by the theorem prover during symbolic execution, when they are set to different values and, by that, all possible variants are simulated.

Example 3.11.
We show an example with our software product line BankAccount, whose feature model is shown in Figure 2.1. The class `FeatureModel` *is shown in Listing 3.8.*

```
package FM;
/**Variability encoding of the feature model for KeY.
 * Automatically generated class by FeatureHouse.
 */
public class FeatureModel {
    public static boolean Interest;
    public static boolean Overdraft;
    public static boolean Logging;
    public static boolean InterestEstimation;
    public static boolean TransactionLog;
    public static boolean Transaction;
    public static boolean DailyLimit;
    public static boolean CreditWorthiness;
    public static boolean BankAccount;
    public static boolean Lock;
}
```

Listing 3.8: Class *FeatureModel* in Metaproduct

Second, we adopt the generation of fields in the metaproduct. Fields can be defined by multiple features. During feature composition, the last feature's definition sets the actual value in the generated product. However, this is not possible for the metaproduct because the variability of the whole product line must be preserved. Hence, [Thüm et al., 2012a] proposed to use the ternary operator to achieve the variability-aware field definition.

Meinicke [2013] noted that a field is not necessarily defined in every configuration. This behaviour may occur when the field is introduced in a refining feature. He proposed to set the field's value to a default value for primitive types (for example `0` for **`int`**) or set to **`null`** for reference types for the respective configurations.

Example 3.12.
We show an example of field generation for metaproducts in Listing 2.2 on Page 9 and Listing 3.9. Both listings contain role Account, Listing 2.2 in feature BankAccount and Listing 3.9 in feature Overdraft. Both roles define a field `OVERDRAFT_LIMIT`. In feature BankAccount, the field's value is set to `0`, but in feature Overdraft it is set to `−5000`. BankAccount is the root feature that is always activated. Therefore the ternary operator checks whether feature Overdraft is activated and sets the value of `OVERDRAFT_LIMIT` accordingly. The result is shown in Listing 3.10 in Line 4. We refer to the listing again below and explain its other parts.

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

Listing 3.9: Role *Account* of Feature *Overdraft*

```
1  public  class Account {
2      public invariant $ValidConfig;
```

```

    public final int OVERDRAFT_LIMIT  = FM.FeatureModel.Overdraft ?
         -5000 :  0;
    public int balance = 0;

    /*@ requires_abs AccountR;
      @ def AccountR = true;
      @ ensures_abs AccountE;
      @ def AccountE = balance == 0;
      @ assignable_abs AccountA;
      @ def AccountA = \nothing;@*/
    Account() {}

    /*@ requires_abs update_BankAccountR;
      @ def update_BankAccountR = FM.FeatureModel.BankAccount && x != 0;
     @ ensures_abs update_BankAccountE;
     @ def update_BankAccountE = (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x);
     @ assignable_abs update_BankAccountA;
     @ def update_BankAccountA = balance; @*/
     private boolean  update_BankAccount  (int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = balance + x;
        return true;
    }
    {...]
    private boolean dispatch_update_DailyLimit(int x) {
        if (FM.FeatureModel.DailyLimit)
            return update_DailyLimit(x);
        return update_BankAccount(x);
    }

    /*@ requires_abs update_DailyLimitR;
      @ def update_DailyLimitR = FM.FeatureModel.DailyLimit && x != 0;
      @ ensures_abs update_DailyLimitE = ((!\result ==> withdraw ==
      @ \old(withdraw))
      @   && (\result ==> withdraw <= \old(withdraw)))
      @ && (!\result ==> balance == \old(balance))
      @   && (\result ==> balance == \old(balance) + x);
      @ assignable_abs update_DailyLimitA;
      @ def update_DailyLimitA; = withdraw, balance; @*/
    private boolean update_DailyLimit(int x) {
        int newWithdraw = withdraw;
        if (x < 0)  {
            newWithdraw += x;
            if (newWithdraw < DAILY_LIMIT)
                return false;
        }
        if (!update_BankAccount(x))
            return false;
        withdraw = newWithdraw;
        return true;
    }
    [...]
    /*@ requires_abs updateR;
      @ def updateR = (FM.FeatureModel.BankAccount
      @ FM.FeatureModel.DailyLimit  FM.FeatureModel.Logging)
      @ && x != 0;
      @ ensures_abs (FM.FeatureModel.BankAccount ==> (!\result ==>
      @ balance == \old(balance))
      @   && (\result ==> balance == \old(balance) + x))
      @ && (FM.FeatureModel.DailyLimit ==> (!\result ==>
      @     withdraw == \old(withdraw))
      @   && (\result ==> withdraw <= \old(withdraw)))
      @ && (FM.FeatureModel.Logging ==> \result ==>
      @     this.updates[this.updateCounter] == x)
      @ && (FM.FeatureModel.Logging ==> \result ==>
      @     this.updateCounter == ( \old(this.updateCounter) + 1 ) % 10)
      @ && (FM.FeatureModel.Logging ==> !\result ==>
      @     this.updateCounter == \old(this.updateCounter));
      @ assignable_abs updateA;
      @ def updateA = withdraw, balance, updateCounter, updates[*];
      @*/
    boolean update(int x) {
        if (FM.FeatureModel.Logging)
            return update_Logging(x);
        return dispatch_update_DailyLimit(x);
    }
    [...]
}
```

Listing 3.10: Class *Account* in Metaproduct

Differences to Meinicke [2013]

Regarding method generation for the metaproduct, our approach differs slightly from Meinicke [2013]. He proposed a generation based on Thüm et al. [2012a]. At the beginning of each method, there is an if-statement checking if the feature this refinement belongs to is selected. This approach is called dynamic branching [Thüm et al., 2012a]. If the corresponding feature is not selected, the next previous implementation of the method is called. This approach, however, leads to a change in our method implementation and the *partial proofs* could not be reused for the metaproduct.

Therefore, we choose a different approach. This approach was introduced by Apel et al. [2013e]. In their work, they used it for variability encoding of methods in software product lines. In the approach, the metaproduct contains two kinds of methods. We call them *dispatcher methods* and *domain methods* for an easier distinction. *Domain methods* represent the methods as they were the feature modules. *Dispatcher methods*, however, dispatch between the different implementations of the *domain methods* of different features [Apel et al., 2013e]. Therefore, a *dispatcher method* for each refinement is introduced. The parameters of the *dispatcher method* are equal to the method that it dispatches to because the parameters need to be propagated. The `dispatcher method` deals with the check whether the corresponding feature is selected. If it is selected, the *domain method* is called. However, if the feature is not selected, either the *dispatcher method* for the next previous method or the introductory implementation is called [Apel et al., 2013e]. If a method is only introduced in one feature, but never refined by another feature, we do not need a dispatcher method because there are no refinements to dispatch between. Additionally, the metaspecification and the contracts of *domain methods* guarantee the necessary variability. In Section 3.2.2, we discuss the specification in more detail.

We adopt this mechanism because we can make sure the implementation of the *domain methods* is not changed, so that we can reuse the *partial proofs* obtained in the feature-based phase. We extend both the *domain methods'* and the *dispatcher methods'* names to indicate the feature they belong to. Additionally, the *dispatcher methods'* names start with the keyword `dispatch` to indicate their purpose. The *dispatcher method* of a method's last refinement keeps the original name of the method without any changes.

Example 3.13.
We present an example of how we create methods in our metaproduct in Listing 3.10. The listing shows the class `Account` of the metaproduct of our BankAccount SPL in part and displays the result of method generation for method `update`. The method is originally defined in the features BankAccount, DailyLimit and Logging. The original implementations of the method are shown in Listing 2.2 (see Page 9) in Line 23 for the feature BankAccount and Listing 2.3 (see Page 9) in Line 46 for the feature DailyLimit. The implemenation of FeatureLogging is not shown for brevity. The dispatcher methods for DailyLimit and Logging are in Line 31 and Line 78 accordingly. Logging is the last feature that refines the method and therefore its dispatcher method is named `update`.

We show the calling hierarchy in Figure 3.1. When method `update` is called, the dispatcher method for feature Logging checks whether the feature is selected. If it is selected, the refinement of Logging is called, otherwise the dispatcher method for the next previous refinement is called. In this case, the next previous refinement is delivered in DailyLimit. If feature DailyLimit is selected, the dispatcher method of DailyLimit calls the corresponding domain method, otherwise the implementation of the last feature BankAccount is called.

Meinicke [2013] discussed several possibilities to generate constructors for a metaproduct. He suggests two approaches (see Meinicke [2013] for a detailed discussion). Both these approaches require that all refinements of a contructor are merged into one constructor. However, when all constructor refinements are merged into one constructor, their contracts are merged into a metaspecification for the resulting constructor as well. The metaspecification is different to each of the individual refinements' contracts and, hence, we cannot reuse the *partial proofs* from the feature-based phase. To solve this issue, we propose a new approach.

In our approach, every constructor refinement gets its own method. Each method only includes the implementation of its own refinement. The methods are named `init`,

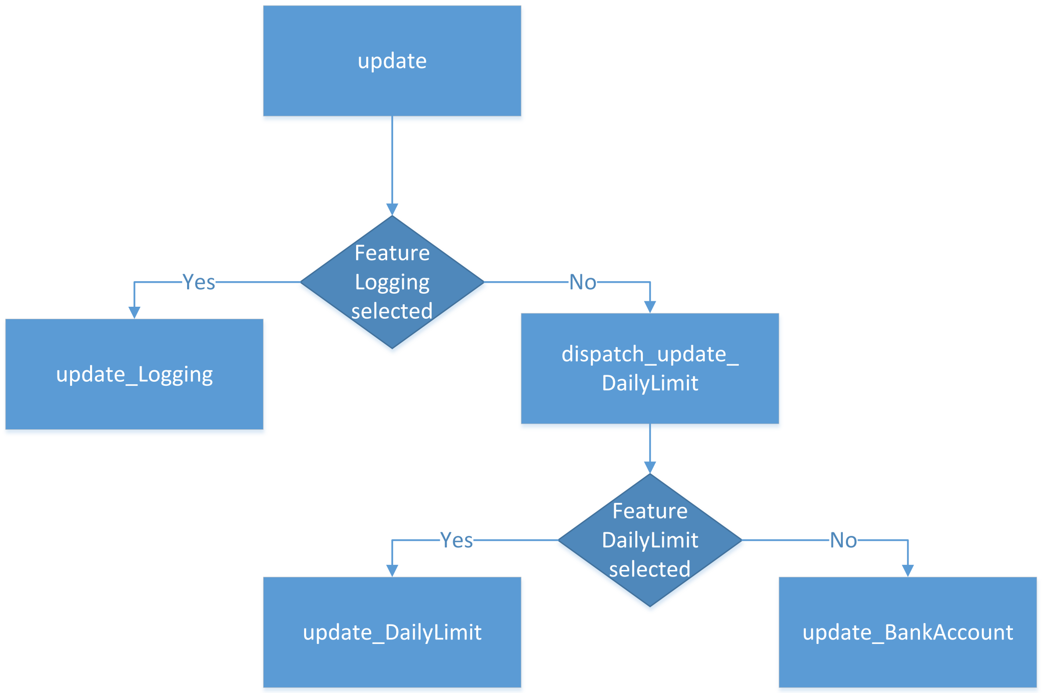


Figure 3.1: Method Calling Hierarchy in Metaproduct

extended by the feature name the constructor comes from (e.g. `init_DailyLimit` for the constructor from feature *DailyLimit*). The actual constructor only contains if-statements checking for the selected features, making the constructor work like a `dispatcher method`. The if-statements are sorted according to the feature composition order. If a feature is selected, its refinement of the constructor is called, otherwise the next check is performed.

Example 3.14.

We continue with a small example of how we generate constructors for our examples. In Listing 3.11, we show the introductory implementation of role Money in feature A. Listing 3.12 shows the refinement of B. Both implementations contain a constructor. When creating our metaproduct, we need to transform these constructors in accordance to our approach. We show the result of the generation in Listing 3.13. For both the introductory and the refining implementation a method `init` *is created that contain the actual code of the constructor. In the constructor itself, both* `init` *methods are called. All three listings also contain the contracts of the constructor. We postpone the discussion about the metaspecification for constructors to Section 3.2.2.*

```
class Money {
    private int amount;

    /*@ ensures amount >=0;
```

```
      @ assignable amount; @*/
    public Money() {
        amount = 0;
    }
}
```

Listing 3.11: Role *Money* of Feature *A*

```
class Money {
    private boolean inDebt;

    /*@ ensures \original;
     @ ensures inDebt == false;
     @ assignable inDebt;@*/
    public Money() {
        inDebt = false;
    }
}
```

Listing 3.12: Role *Money* of Feature *B*

```
class Money {
    private int amount;
    private boolean inDebt;

    /*@ requires_abs MoneyR;
      @ def MoneyR = FM.FeatureModel.A &&
      @ (! FM.FeatureModel.B  FM.FeatureModel.A);
      @ ensures_abs MoneyE
      @ def MoneyE = (amount >= 0)
      @ && (FM.FeatureModel.B ==> (inDebt == false));
      @ assignabel_abs MoneyA;
      @ def MoneyA = amount, inDebt;
      @*/
      @ assignable amount, inDebt; @*/
    public Money() {
        if (FM.FeatureModel.A) {
            init_A();
        }
        if (FM.FeatureModel.B) {
            init_B();
        }
    }

    /*@ requires_abs init_AR;
      @ def init_AR = true;
      @ ensures_abs init_AE
      @ def init_AE = (amount >= 0);
      @ assignabel_abs init_AA;
      @ def init_AA = amount; @*/
    public /*@ helper @*/ init_A() {
        amount = 0;
    }

    /*@ requires_abs init_BR;
      @ def init_BR = true;
      @ ensures_abs init_BE
      @ def init_BE = inDebt == false;
      @ assignable_abs init_BA;
      @ def init_BA = inDebt; @*/
    public /*@ helper @*/ init_B() {
        inDebt = false;
    }
}
```

Listing 3.13: Class *Money* in Metaproduct

3.2.2 Generation of the Metaspecification

Again, we base our algorithm on the work of Thüm et al. [2012a] and Meinicke [2013]. However, because of our changes in Section 3.2.1, there also are multiple changes here, which is why we do not differentiate between commonalities and differences.

As with the feature stubs, we transform all concrete method contracts into abstract method contracts. We do this to be able to reapply the proofs obtained in the feature-based verification. Again, we define the abstract sections for **requires**, **ensures**, and **assignable** clauses and declare the placeholders. However, the content of the placeholder definition depends on the kind of method.

We need to consider three kinds of methods for the contract generation for the metaproduct. First, there are the contracts for the *domain methods*. Their concrete sections are exactly as in the feature stubs because their implementation is not changed either and the *partial proofs* from the previous phase are to be reused on them. Therefore, for contracts with multiple clauses of the same type, we also compose them into one clause by means of logical And. Furthermore, we also add the **requires** clause stating the feature that needs to be active in order to be able to call this *domain method*. To be able to reapply the *partial proofs*, we must either name the placeholders exactly like in the feature-based phase or update the placeholder names in the *partial proofs*. For *domain methods*, whose contract's concrete section was removed in the feature-based phase (see Section 3.1.2), we now apply the same procedure as we do for the other methods.

Additionally, we need to deal with the keyword **original** in *domain methods*' contracts. In Section 3.1.2, we discussed three different approaches to how to deal with the keyword in the feature-based phase. Regardless of which approach is used, we can replace the keyword with concrete clauses of the previous refinements. If all contract refinements include this keyword, we can compose them along the complete calling hierarchy. However, if the hierarchy is incomplete, we can use feature variables to indicate, which clauses need to be fulfilled for a given feature combination.

Example 3.15.
We show an example for the contract generation of domain methods in Listing 3.10. The

*listing includes method `update_DailyLimit`. This method is originally defined in role Account of feature DailyLimit, which can be seen in Listing 2.3 on Page 9. In accordance to our proposed approach, the contracts of `update` is transformed into an abstract contract for `update_DailyLimit` and the content of the different clauses is adopted. Additionally, the **`requires`** clause `FM.FeatureModel.DailyLimit` is added to indicate that the feature variable of feature DailyLimit must be **`true`** in order to call the method. Furthermore, the placeholder names are equal to the placeholder name in the respective feature stub, so that the corresponding partial proof from the feature-based verification can be re-used. Only the keywords **`original`** in the **`requires`** and **`ensures`** clauses are replaced by the pre- and postcondition of `update_BankAccount`.*

The second kind of contracts covers *dispatcher methods*. Their contract is equal to the contract proposed by Meinicke [2013] for methods in his metaproduct, except that we generate them as abstract contracts. Therefore, we only give a brief summary of the contracts' structure. For a more detailed discussion, see Meinicke [2013]. For the concrete sections of *dispatcher methods*' contracts, we compose the contracts from *domain methods* previous to the `dispatcher method`. The contract works similar to the *dispatcher method* itself. In front of every individual **`requires`** or **`ensures`** clause, obtained from one of *domain methods*, an implication states, which feature needs to be selected, so that the respective clause needs to be fulfilled. Meinicke [2013] also proposed to add a **`requires`** clause defining the minimal feature selection for the method to be callable. We also adopt this proposal and add this **`requires`** clause to the concrete section of each *dispatcher method*'s contract. Finally, the **`assignable`** clauses of the different contract refinements are composed as well. There is not yet an established way of handling **`assignable`** clauses in feature-oriented programming. For practicality reasons, we assume, that it is possible to merge the **`assignable`**clauses of all contract refinements into one **`assignable`** clause. However, our approach to **`assignable`** of *dispatcher methods* may be up for revision if our assumption is wrong.

Example 3.16.
*In Listing 3.10, we present an example for illustration. We transformed the contract of method `update` into an abstract contract. The contract contains all pre- and postconditions of the domain methods that may get called starting from this method. However, for every pre- and postcondition, there is a check for the respectively selected feature. Additionally, we add the **`requires`** clause `FM.FeatureModel.BankAccount || FM.FeatureModel.DailyLimit || FM.FeatureModel.Logging` as the method is implemented in these three features.*

Third, we need to generate contracts for constructors. As the actual constructor serves as a *dispatcher method* for all constructor refinements, its contracts are only a special case for *dispatcher methods*, which we discussed above. The contracts for the different `init` methods, on the other hand, represent a special case of contracts for *domain methods*. We also discussed them above. However, we add the keyword **`helper`** to `init` methods' contracts. This keyword is used to liberate a method from the obligation to fulfil the classes invariants[Leavens et al., 2008]. We use the keyword for `init` methods because they are actually part of the constructor, which serves to first establish all invariants upon its completion. Therefore, a specific `init` method may not be able to fulfil all invariants.

Example 3.17.
We give a small example to illustrate the contract generation for contructors. We refer, again, to Listing 3.11, Listing 3.12 and Listing 3.13, which we used above to illustrate the generation of constructors in the metaproduct. In the first two listings, we show the contracts of the two constructor implementation. They are composed in accordance to our proposed approach and we show the result in the third listing. In the metaproduct, we enrich both `init methods` *with the contracts of the constructors from the original features. Additionally, we add the keyword* **`helper`** *to both* `init` *methods. For the actual constructor, a metaspecification is created according to how contracts for dispatcher methods are created.*

To ensure, that only valid configurations are used by the theorem prover, we add an invariant to every class stating the feature model as a first order propositional formula. If a configuration is not valid, methods do not need to fulfil their contract. We do not include this validation into each method contract because it the tested configuration must be valid both before and after a method execution. Therefore, if we added the validation checks to the method contracts, they would make the contracts bigger than necessary. Additionally, using an invariant for the configuration validation allows us to omit *dispatcher methods* for methods that are only introduced in one feature, but never refined in another feature, as `domain methods` cannot include variability information.

Example 3.18.
We show this invariant for the BankAccount SPL in Listing 3.10 in Line 2. We replaced the boolean expression in the listing for readability reasons. It is semantically equal to the expression in Listing 2.1 on Page 7.

Finally, we need to transform invariants from the feature modules for the metaproduct. As with individual clauses of constracts of *dispatcher methods*, invariants only have to hold under specific feature combinations. Meinicke [2013] proposed to use an implication again to indicate, which features need to be selected. We adopt this mechanism.

3.2.3 Adaption of the Partial Proofs

After the creation of the metaproduct, we need to adapt the *partial proofs* from the feature-based verification to be actually able to reuse them for the metaproduct. This mainly includes the updating of names of elements from feature stubs to meta product because the method contracts and implementation need to match. As field and class names do not change either in the generation of the feature stubs or the generation of the metaproduct, their names are already correct in the *partial proofs*. However, the names of methods do change in both generation processes. Additionally, if the placholders' in the *partial proof* do not match the placeholders' name in the metaproduct, we need to update these, as well. Therefore, we discuss the necessary renaming in more detail.

First, all saved *partial proofs* contain the name of the method whose *partial proof* they represent. We call that name *proved method name*. In the feature stubs, the method names are the original names. In the metaproduct, the *domain methods* are renamed in accordance to what we described in Section 3.2.1. We want to reuse the *partial proofs* for the *domain methods*, we obtained in the feature-based phase, in this phase. Therefore, we need to change the proved method names in the *partial proofs* accordingly.

Example 3.19.
We show an example to further illustrate which changes need to be made in the partial proofs obtained in the feature-based verification. Listing 2.3 shows role Account for feature DailyLimit. We focus on method `update` for this example. In the corresponding feature stub for this role, which can be seen in Listing 3.3, the method name is `update`. In contrast, in the metaproduct, which can be seen in Listing 3.10, the name is changed to `update_DailyLimit`. Hence, the proved method name in the according partial proof from the feature-based phase needs to be changed to `update_DailyLimit` to be reusable for the metaproduct.

Second, methods can call previous implementations by means of the keyword **`original`**. The feature stubs represent this mechanism by a prototype that is added to the class and called instead of the keyword **`original`**. In the metaproduct, this mechanism is realized by replacing the keyword **`original`** with a call to the *dispatcher method* of the previous refinement.

Example 3.20.

We continue with Example 3.19. Method `update` in role Account of feature DailyLimit uses the keyword **`original`** *to call a previous implementation. In its respective feature stub, the keyword was replaced by a call to the method prototype `update_original_DailyLimit` (see Line 10 in Listing 3.3). In Listing 3.10, we show the resulting metaproduct. In the metaproduct, the keyword is replaced by a call to method `update_BankAccount`. This method `update_BankAccount` is the introduction of the method. Thus, the call to `update_original_DailyLimit` in the partial proof is replaced by a call to method `update_BankAccount` and the placeholders of `update_original_DailyLimit` are replaced by the placeholders of `update_BankAccount`.*

We apply a similar procedure to calls to methods of other features. We do not need to change the method names, as both in the feature stubs and in the metaproduct, the actual calls are not changed. In the feature stubs, for each of these calls a method prototype is generated to match the call and avoid type errors. However, we need to update the placeholders' names, if they are included into the proof and are different in the metaproduct.

3.2.4 Family-Based Theoreom Proving

In this second phase, we verify the created metaproduct. With a metaproduct, it is possible to simulate all valid configurations at runtime without generating each product on its own. Our goal is to heavily reuse *partial proofs* from phase one because replaying a proof is easier than to find one [Necula, 1997].

To verify all valid configurations, the theorem prover can set all feature variables respectively. By that, all methods can be proven for all valid feature combinations. In this phase, we need to distinguish two different kinds of methods that need to be verified. For the *domain methods*, we can reuse the *partial proofs*, that were the result of the feature-based theorem proving. If all proof goals can be closed, the proof is accomplished. However, it is possible that by replaying a method's *partial proof* not all proof goals can be closed (see Section 3.1.3). If replaying does not close all proof goals, the theorem prover is used to prove the rest of the goals as in this phase all contracts are concrete and every goal should be closable. Second, we need to verify the *dispatcher methods*. We do not have any information about them from earlier phases of our theorem proving. Therefore, we employ the theorem prover to verify them.

We can verify methods either by inlining the body of all called methods or by relying on the contract of the called method [Beckert et al., 2007]. If we inline the method bodies, it is only necessary to verify the last dispatcher method because it includes all variability information of the *dispatcher methods* that may be called within it. By doing that, we do, however, not profit from the feature-based phase at all because we do not need any of the *partial proofs* from the *domain methods*. Hence, for our approach, relying on the contracts is more promising. With this approach, we need to prove all *dispatcher methods* and *domain methods*. However, we can reuse the *partial proofs* from the feature-based phase and the *dispatcher methods* are less complex and long.

If the theorem prover still cannot close all goals either contract or implementation do not match or the theorem prover is not able to verify the method(see discussion in Section 3.1.3). However, due to the structure of the metaproduct, it is not easily possible to know which feature combination causes the problem [Meinicke, 2013]. *Domain methods* can be traced back to their features. Hence, we can, in that case, at least suspect the feature, in which the problem occurs. For *dispatcher methods*, the approach is similar. When a proof of such a method cannot be closed, we can try the next previous `dispatcher method`. If that previous `dispatcher method` can be verified, we can suspect the problem to be in the refinement.

3.3 Summary

In this chapter, we presented our approaches to feature-based and feature-family-based theorem proving for software product lines. Our motivation for the development of these approaches based on the inherent limitations of feature-based and family-based analysis approaches for software product lines. While feature-based strategies are necessarily insufficient because they only consider features in isolation, family-based verification is both consumptive of resources for large software product lines and needs to be re-performed whenever code changes. Thus, the goal was to combine these approaches to overcome these shortcomings and embrace existing benefits.

In order to achieve such a combination, we first had to develop a feature-based approach, whose results could be reused later in the feature-family-based strategy. Feature-based theorem proving focuses on the verification of features. As features are not compilable programs and the behaviour of not compilable programs is not defined, we employ the concept of feature stubs. Feature stubs provide extensions to feature modules to transform them into valid Java programs. To realize these extensions, abstract contracts are used to map contracts of methods that are called in a feature, in which they are not defined. These feature stubs can be verified with a regular theorem prover. The verification results, which consist of a *partial proof* for each method defined within the product line that has a method contract, are then saved for potential reuse in the second phase.

To reuse these *partial proofs* in the family-based verification phase, we designed a metaproduct based on the work of Thüm et al. [2012a] and Meinicke [2013], whose structure is compatible with the *partial proof*. There are two method types in the metaproduct, *dispatcher methods* and *domain methods*. The first kind is used to deal with the different refinements in different feature and dispatch between them. Due to their function, they are completely verified in this phase. The *domain methods* represent the actual refinements. On these methods, the *partial proofs* can be replayed. If all proof goals can be closed, the software product line is verified successfully.

4. Implementation

In the last chapter, we discussed our concept of a feature-family-based verification of software product lines. Our main contribution was the first phase as a feature-based verification strategy whose results can later be reused. This strategy includes a mechanism to generate feature stubs that allow for a feature-based verification and the verification of these feature stubs. As this generation includes several extensions to the existing feature modules, it requires expertise and is time consuming. We implemented a tool to mostly automate this generation and present the tool in this chapter.

We separate our main tasks feature stub generation and theorem proving on the tools FeatureHouse, FeatureIDE and KeY, respectively. We not only aim to realize both functionalities but also to provide a cooperation of the tools for a better user experience. We use the existing tool support provided by FeatureIDE and integrate our feature stub generation. FeatureIDE and KeY cooperate in terms of an automated transition between feature stub generation and verification. This automatism also required small changes in KeY.

We give an overview of the tool support that is required to implement our concept in refsec:req-tool-support. In Section 4.2, we give an overview about the tools we extended for a better support of our concept. We briefly describe their functionality and what they are used for. Section 4.3 provides information about how we integrated our feature-stub generation algorithm in FeatureIDE and how the algorithm is implemented. Our extensions to KeY are explained in Section 4.4. In Section 4.5, we conclude the chapter with limitations of our implementation.

4.1 Requirements for Tool Support

Our concept is a feature-family-based approach to software product-line verification by means of abstract contracts. The tool support must therefore support these concepts.

We need both a development tool suitable for feature-oriented programming and a verification tool that is able to handle abstract contracts.

In particular, the development tool needs to support the generation of feature stubs and the metaproduct. The development tool must be suitable for feature-oriented programming because it must recognize features and *feature module dependencies* for the generation of the feature stubs. Additionally, our tool must be able to perform all checks and changes to the original feature module discussed in Section 3.1.1 and Section 3.1.2. These checks and changes include a type check, the collection, and analysis of access information, as well as the generation of classes, fields, methods, and method contracts. We require the collected access information to include both accesses of fields, methods, and types from within the source code and from contracts. For the metaproduct, our tool needs to handle the variability information of a software product line and create methods, fields, and contracts according to sections 3.2.1 and 3.2.2.

The verification tool must be able to perform our feature-based and family-based verifications. We design both the feature stubs and the metaproduct so that they are valid Java programs with JML method contracts and invariants. Hence, a suitable verification tool needs to perform theorem proving on these programs. For the feature-based verification, we require the tool to support reasoning with abstract contract as well as creating and saving *partial proofs* for a method. For the family-based verification, our verification tool needs to support proof replay.

4.2 Existing Tool Support

In the last part, we discussed requirements for the implementation of our concept. In this part, we present tools, which we use and extend in order to realize this implementation. Thereby, we both give a small introduction to the tool and address how they can help us in realizing the necessary tool support.

FeatureIDE is an integrated development environment for the development of software product lines and supports all phases of software product line engineering [Thüm et al., 2014b]. Its focus is feature-oriented programming but is not limited to that. FeatureIDE supports several composition tools such as FeatureHouse for feature-oriented programming in Java, C# or Haskell, AHEAD for feature-oriented programming in Java, and FeatureC++ for feature-oriented programming in C++. Different preprocessor implementations such as Munge for Java and Android, Antenna for Java, and DeltaJ for delta-oriented programming are also supported. FeatureIDE is a plug-in for the Eclipse framework and is open source [Thüm et al., 2014b]. We use FeatureIDE for general handling of software product lines because of its advanced support for Java-based software product lines. It also supports the creation and management of JML contracts [Benduhn, 2012; Proksch and Krüger, 2014]. With FeatureIDE, we can also generate the necessary classes, methods, fields and contracts.

FeatureHouse is an open-source language-independent framework for the composition of software artefacts [Apel et al., 2013b]. It is employed by FeatureIDE to implement feature composition. Internally, FeatureHouse uses feature structure trees (FST) that are generated from feature modules [Apel et al., 2013b]. The FSTs are used tor implement superimposition. Besides composition of source code, FeatureHouse additionally supports JML-based contracts and several mechanisms for their composition [Benduhn, 2012]. We use FeatureHouse as a composer because, among the composers integrated in FeatureIDE, it is the only one support JML contracts.

Fuji is a compiler for Java-based feature-oriented programming [Apel et al., 2012]. Its syntax is based on FeatureHouse and Jak and it supports superimposition and product generation. Fuji goes beyond regular composition tools and directly creates Java byte code [Apel et al., 2012]. Fuji was created as an implementation of an access modifier model proposed by [Apel et al., 2012] to tackle the issue of access control in feature-oriented programming. Due to its extended functionalities, Fuji cannot only be used for regular product generation but also for a family-based type checking. With its access model, Fuji can provide information about accessed methods, fields and types inside the Software Product Line. Furthermore, we can employ Fuji for the type check of the software product line before creating the feature stubs.

KeY is a software analysis tool for Java [Ahrendt et al., 2014]. Its core functionality is semi-automatic theorem proving of regular object-oriented programs by deductive verification [Beckert et al., 2007]. It supports source code conforming to Java Card and specification defined by JML[Ahrendt et al., 2014]. KeY transforms contracts into first order dynamic logic and uses symbolic execution to prove a method behaves accordingly [Ahrendt et al., 2014]. Besides theorem proving, the KeY platform provides other analyses such as information flow analysis which are implemented by different editions in the KeY project. KeY is available as a stand-alone and an Eclipse project. We use KeY in version 2.1 with an extension that integrates abstract contracts. We use KeY because it can provide theorem proving with abstract contracts. Additionally, we can use KeY for saving *partial proofs* and replaying existing proofs.

4.3 Generation of Feature Stubs

The feature stub generation is accessible in FeatureIDE as an option in the context menu of FeatureHouse projects. We show a screenshot of the context menu in Figure 4.1. Our implementation focusses on FeatureHouse because it is the only composition tool integrated into FeatureIDE.

First, we build the FST model because it provides us information about the structure of the software product line, its methods, fields, and contracts. We use this information for the feature stub generation. Then, we employ the compiler Fuji to both perform a family-based type check. If an error occurs during the type check, the verification is aborted because a non-compilable program's behaviour is not defined. If the type check is successful, we use Fuji again to obtain all access information for the software product line to determine the fields, types and methods each method accesses. With this information, we can perform the checks and additions for the feature stubs, we discussed in Section 3.1.1. Below, we describe their implementation in more detail.

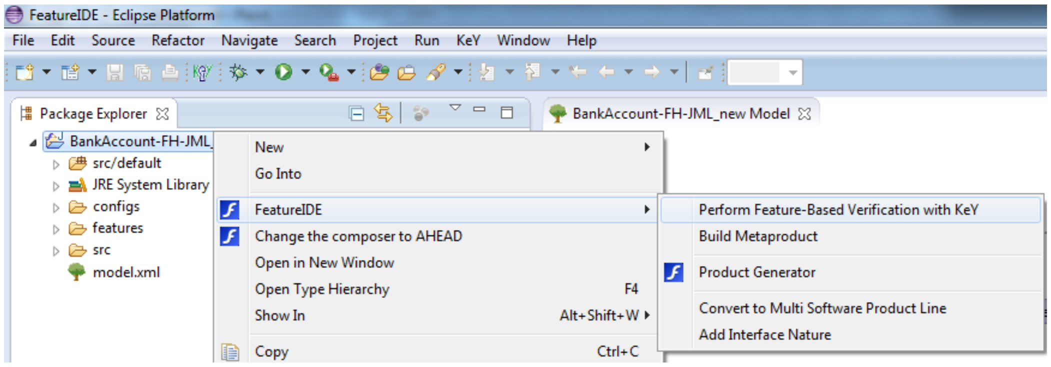

FeatureIDE - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run KeY Window Help
Package Explorer
BankAccount-FH-JML_new Model
BankAccount-FH-JML
src/default
JRE System Library
configs
features
src
model.xml
New
Go Into
FeatureIDE
Change the composer to AHEAD
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Perform Feature-Based Verification with KeY
Build Metaproduct
Product Generator
Convert to Multi Software Product Line
Add Interface Nature

Figure 4.1: Screenshot of Context Menu Entry For Activation of Feature Stub Generation

We create the feature stubs in a designated subfolder of the feature project's root folder. For each feature stub a folder is created inside the feature subfolder. All roles are copied from the feature folders into the feature stub subfolder and we create the `FeatureModel` class, including the respective feature variable as a boolean field. Every time the feature stub generation is started all feature folders in the feature stub folder are deleted.

To create all additions, we consecutively check for all cases, discussed in Section 3.1.1. As the checks and generation take some time, we encapsulated them into their own thread. By that, it is possible to continue using FeatureIDE while the feature stub generation and verification are running in the background. Each feature module gets individually transformed into a feature stub and proven afterwards. Only after the verification of one feature is performed the feature stub generation for the next feature begins.

For the checks, we iterate through all methods of a feature provided by the FST model and determine the elements it accesses. Each of the check is implemented by an if-statement, testing for the respective condition. We first check for external methods. If a method is used that is not part of the software product line, we insert a message into the error log and the feature stub generation is stopped. For further clarification, the message includes method name and class.

Then, we check for calls to previous implementations by means of the keyword **`original`** in methods. In accordance with our proposed procedure in Section 3.1.1, we create a method prototype and replace the keyword by a call to that prototype. Additionally, we generate the abstract section of an abstract contract for that method prototype.

We also perform a check for the keyword **`original`** in contracts. This keyword can be used in explicit contract refinement to include the contract of a previous refinement of the method. In Section 3.1.1, we discussed several alternatives to deal with the keyword, as we ignore other features and the variability model of the software product line. We replace the concrete section of the contract by the abstract section of the contract, containing only the declaration of the placeholders. This choice allows us to at least create a *partial proof* for the abstract section.

Our next check is the check for missing types. We use the type information, obtained by Fuji, to find accessed types. If an accessed type is not defined within the feature, a class prototype is created in the respective feature folder in the feature stub folder.

We check for accessed methods and fields that are not defined within the feature, the method accessing them belongs to. In accordance to our proposed approach in Section 3.1.1 and Section 3.1.2, we create method and field prototypes. The prototypes are accompanied by a comment indicating that they are prototypes to simplify the distinction between regular fields and methods from the feature module and these added for the feature stub. Additionally, we create abstract sections of abstract method contracts for the method prototypes, so that they are represented in the verification.

We include a check if a feature even contains a JML contract at all. This check was motivated by our running example *BankAccount SPL* because it contains the feature *Overdraft* which does not specify any contracts. If a feature does not contain any contracts, it is simply skipped for the feature-based verification because there is nothing to verify.

After all checks are performed on a method, we transform the contract of that method into an abstract contract. We use the concrete contract as the definition of the placeholders. Furthermore, we add the **`requires`** clause stating that the respective feature variable in the `FeatureModel` class is **`true`**.

Our tool distributes the tasks described above to three classes. In the class diagram Figure 4.2, we show the classes and how they interact with each other. The first class `GenerateFeatureStubAction` represents an Action that deals with the user input when clicking the context menu entry seen in Figure 4.1. When a user clicks the option to create a project's feature stubs and to prove them afterwards, the class' method `run` is called, creates an object of `FeatureStubGenerator` and starts the generation. The class `FeatureStubGenerator` is the second class in our implementation. It is responsible for the actual generation of the feature stubs, calls Fuji and KeY. We create the feature stubs in an own thread in the method `createFeatureStub` to not prohibit the user from further interacting with FeaturIDE. The checks discussed above are performed in the methods `isInCurrentFeature`, `checkForMissingTypes`, `checkForOriginaInContract`, and `checkForOriginal`. The methods `createClassForPrototype` and `createPrototype` accomplish the generation of the field, class and method prototypes. The third class is `ExtendedFujiSignatures`. In this class, we employ Fuji to collect access information about the software product line. In method `findMethodAccesses`, we examine each methods on the methods, fields and types, it accesses. For each field, method and type, we store from which method they are accessed in method `putAccess`. Metho `createSignatures` collects all access data and returns it to the `FeatureStubGenerator`.

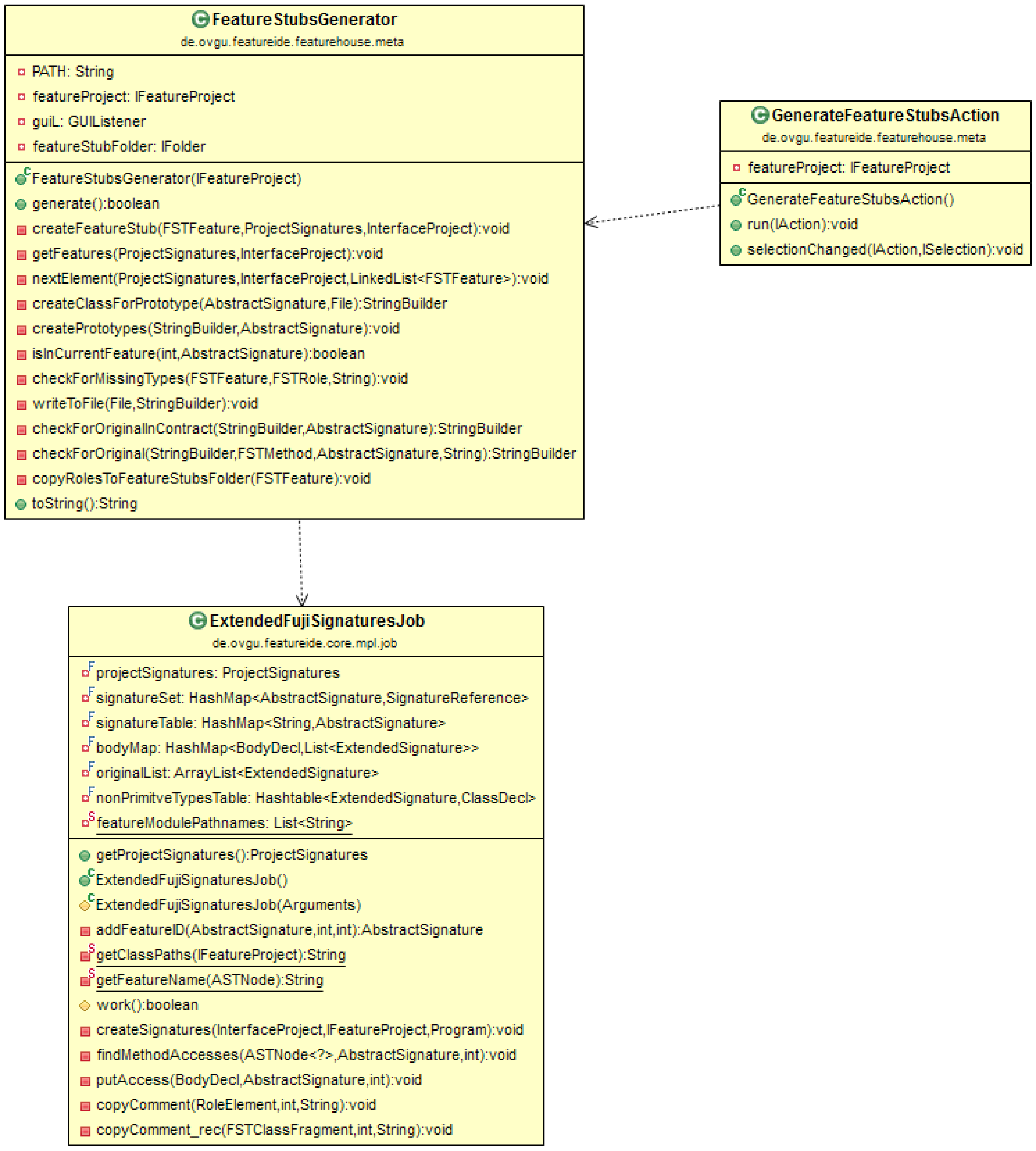
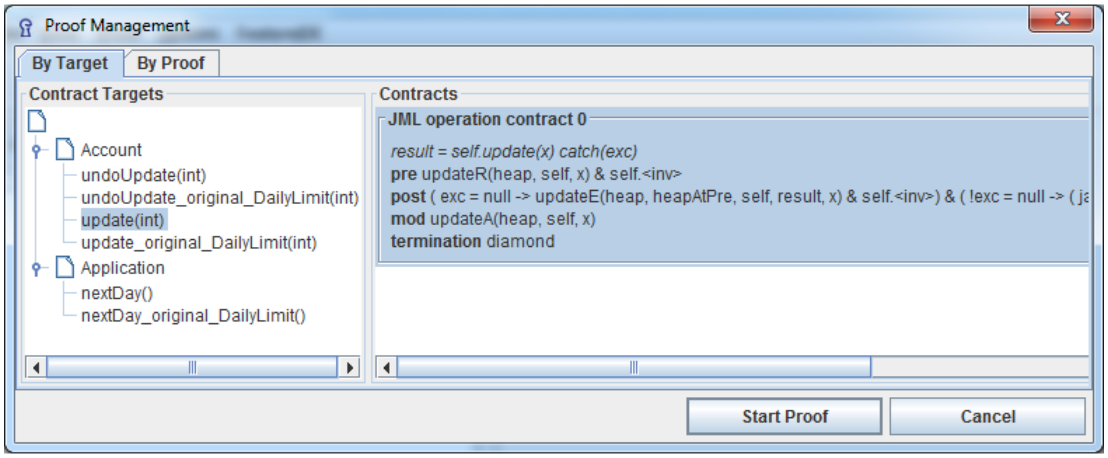

FeatureStubsGenerator
de.ovgu.featureide.featurehouse.meta
PATH: String
featureProject: IFeatureProject
guiL: GUIListener
featureStubFolder: IFolder
FeatureStubsGenerator(IFeatureProject)
generate():boolean
createFeatureStub(FSTFeature,ProjectSignatures,InterfaceProject):void
getFeatures(ProjectSignatures,InterfaceProject):void
nextElement(ProjectSignatures,InterfaceProject,LinkedList<FSTFeature>):void
createClassForPrototype(AbstractSignature,File):StringBuilder
createPrototypes(StringBuilder,AbstractSignature):void
isInCurrentFeature(int,AbstractSignature):boolean
checkForMissingTypes(FSTFeature,FSTRole,String):void
writeToFile(File,StringBuilder):void
checkForOriginalInContract(StringBuilder,AbstractSignature):StringBuilder
checkForOriginal(StringBuilder,FSTMethod,AbstractSignature,String):StringBuilder
copyRolesToFeatureStubsFolder(FSTFeature):void
toString():String
GenerateFeatureStubsAction
de.ovgu.featureide.featurehouse.meta
featureProject: IFeatureProject
GenerateFeatureStubsAction()
run(IAction):void
selectionChanged(IAction,ISelection):void
ExtendedFujiSignaturesJob
de.ovgu.featureide.core.mpl.job
projectSignatures: ProjectSignatures
signatureSet: HashMap<AbstractSignature,SignatureReference>
signatureTable: HashMap<String,AbstractSignature>
bodyMap: HashMap<BodyDecl,List<ExtendedSignature>>
originalList: ArrayList<ExtendedSignature>
nonPrimitveTypesTable: Hashtable<ExtendedSignature,ClassDecl>
featureModulePathnames: List<String>
getProjectSignatures():ProjectSignatures
ExtendedFujiSignaturesJob()
ExtendedFujiSignaturesJob(Arguments)
addFeatureID(AbstractSignature,int,int):AbstractSignature
getClassPaths(IFeatureProject):String
getFeatureName(ASTNode):String
work():boolean
createSignatures(InterfaceProject,IFeatureProject,Program):void
findMethodAccesses(ASTNode<?>,AbstractSignature,int):void
putAccess(BodyDecl,AbstractSignature,int):void
copyComment(RoleElement,int,String):void
copyComment_rec(FSTClassFragment,int,String):void

Figure 4.2: Class Diagram of the Implementation



```
result = self.update(x) catch(exc)
pre updateR(heap, self, x) & self.<inv>
post ( exc = null -> updateE(heap, heapAtPre, self, result, x) & self.<inv>) & ( !exc = null -> ( ja
mod updateA(heap, self, x)
termination diamond
```

Figure 4.3: KeY Proof Management

4.4 Feature-Based Theorem Proving With KeY

We use KeY as an Eclipse plugin because it is the easiest way to establish a cooperation between FeatureIDE and KeY. This way, both plugins can be installed individually to Eclipse and interact when both are available.

After the generation of a feature stub, we automatically start KeY with the feature stub as input. In Figure 4.3, we show the proof management of KeY for the feature stub of *DailyLimit*. We did not need to change the verification process because the feature stubs are designed to be verified with a theorem prover for regular object-oriented programs. After verification, the *partial proofs* are saved with KeY and it can be closed.

We aim to automate the loading process of the next feature stub to provide a more fluent transition between the tools. Therefore, we make some changes in KeY. KeY's user interface provides a `GUIListener` that notifies when KeY is closed. We change the user interface's modifier to **`public`**, so that FeatureIDE can access it and implement the `GUIListener`. Additionally, we add a constructor for KeY's `Main` class and created a new *init* method for the class. The `init` method contains the initialization of KeY. It es called either by the new constructor or the old `main`, depending on whether KeY is started as an individual program or as a plugin from FeatureIDE. To better distinguish these two cases, we added a second parameter to the `init` method indicating the way, in which KeY was started.

In KeY, we use the macro *"Finish abstract proof part"* to reason about abstract contracts. We use the macro with KeY's option "Use Contracts" to verify the methods of our feature stubs. When the macro has finished, we save the *partial proof* as *proof* files. We show a *proof* file in Listing 4.1 in part. The proof is for method `update` of role *Account* of feature *BankAccount*. The listing shows the first rules KeY applied for proving the method. We leave out most of the strategy settings documented in the beginning of the proof file and only show the start of the proof. We show the method itself in Listing 2.3 on page 9. We have not implemented the *partial proof* adaption.

Nonetheless, we want to demonstrate the changes that need to be made, so that we can replay them for the metaproduct. In Line 6 and Line 7 of Listing 4.1, the *proved method names* are saved. Both occurrences of the name need to be changed before reusing this *partial proof* for our metaproduct of a software product line. The proof starts in Line 11. In Line 33, a previous implementation of method `update` is called. This call represents a *feature module dependency*, as discussed in Chapter 3. In accordance with our concept, the call needs to be changed `update_BankAccount`.

```
 1 [...]
 2 \javaSource "featurestub/DailyLimit";
 3 
 4 \proofObligation "#Proof Obligation Settings
 5 [...]
 6 name=DailyLimit[Account\\:\\:update(int)].JML operation contract.0
 7 contract=DailyLimit[Account\\:\\:update(int)].JML operation contract.0
 8 class=de.uka.ilkd.key.proof.init.FunctionalOperationContractPO
 9 ";
10 [...]
11 \proof {
12 [...]
13 (branch "dummy ID"
14 (builtin "One Step Simplification" (formula "1"))
15 (rule "impRight" (formula "1"))
16 (rule "andLeft" (formula "1"))
17 (rule "andLeft" (formula "2"))
18 (rule "andLeft" (formula "1"))
19 (rule "andLeft" (formula "1"))
20 (rule "andLeft" (formula "1"))
21 (rule "andLeft" (formula "1"))
22 (rule "notLeft" (formula "2"))
23 (rule "eqSymm" (formula "8") (term "1,0,0,1,0,1"))
24 (rule "assignment" (formula "8") (term "1"))
25 (builtin "One Step Simplification" (formula "8"))
26 (rule "methodBodyExpand" (formula "8") (term "1") (newnames
27  "heapBefore_update,savedHeapBefore_update"))
28 [...]
29 (rule "compound_assignment_2" (formula "10") (term "1") (inst "#v=x_9"))
30 (rule "variableDeclarationAssign" (formula "10") (term "1"))
31 (rule "variableDeclaration" (formula "10") (term "1") (newnames "x_9"))
32 (builtin "Use Operation Contract" (formula "10") (newnames
33   "heapBefore_update_original_DailyLimit_0,result_2,exc_2,
34  heapAfter_update_original_DailyLimit_0,
35  anon_heap_update_original_DailyLimit_0") (contract "Account[Account::
36 update_original_DailyLimit(int)].JML normal_behavior operation contract"))
37       (branch "Post (update_original_DailyLimit)"
38    (builtin "One Step Simplification" (formula "9"))
39    (opengoal " ")
40    [...]
41 ))}
```

Listing 4.1: Partial Proof For Method *update* of Role *Account* of Feature *DailyLimit*

4.5 Limitations of the Implementation

Our tool has some limitations compared to our concept presented in Chapter 3. We focussed on implementing the feature-based phase, but did not implement second phase as it would have gone beyond the limits of our thesis.

We use Fuji to gain information about which methods of the software product line access which methods, fields and types. However, Fuji does not consider contracts. This means that if an element is defined in one feature and accessed in another feature but only in one or several of its contracts, Fuji does not notice this access. The prototypes need to be built by hand.

Finally, in our current implementation, we delete all feature folders in the feature stub folder at the beginning of each generation. We do not perform the re-verification check for feature stubs as part of our implementation. The check, however, can be performed manually because only the feature stubs are deleted and newly generated and the proofs are preserved.

4.6 Summary

In this chapter, we presented our implementation of the feature-based verification. Our tool supports the generation of feature stubs as proposed in Chapter 3. Furthermore, we showed, how the development environment FeatureIDE and the verification tool KeY cooperate with each other. To give a the reader a better overview of our implementation, we list all requirements and the respective tool accomplishing the the task in Table 4.1. Our tool enables users to perform a complete feature-based verification of a software product line with a few clicks and makes it possible to obtain *partial proofs* that can be reused in a later family-based verification. Finally, we explored shortcomings of our implementation with respect to our concept.

Table 4.1: Overview of Tool Support

Functionality	Tool
1. Feature-Aware Development of SPLs	FeatureIDE
2. Support for Superimposition	FeatureHouse
3. Support for JML Contracts	FeatureHouse
4. Feature-Based Phase	
4.1 Type Check of Software Product Lines	Fuji
4.2 Access Information	
4.2.1 Access from Source Code	Fuji
4.2.2 Access from Contracts	-
4.3 Generation of Prototypes	Our Extension to FeatureIDE
4.4 Theorem Proving on Valid Java Program	KeY
4.4.1 Reasoning with Abstract Contracts	KeY
4.4.2 Saving Partial Proofs	KeY
4.5 Transition from Feature Stub Generation to Theorem Proving	
4.5.1 Automatic Start of Verification Tool	Our Extension to FeatureIDE
4.5.2 Return to Feature Stub Generation After Verification	Our Extension to KeY
5. Family-Based Phase	
5.1 Generation of Metaproduct	-
5.2 Partial Proof Adaption	-
5.3 Theorem Proving on Valid Java Program	KeY
5.3.1 Proof Replay	KeY

5. Evaluation

In Chapter 3, we presented our concept for a feature-family-based verification of software product lines. We described the implementation of parts of our concept in Chapter 4. To evaluate both our concept and the implementation, we test our approach in comparison to already existing approaches. In this chapter, we present and discuss the results.

In Section 5.1, we present the software product line that we use for our evaluation in more detail. We discuss both relevant aspects of the software product line as well as the different versions, which we use for testing. Then, we describe how the evaluation is performed and which approaches are compared to each other in Section 5.2. In Section 5.3, we present and explain the results and discuss them in Section 5.4.

5.1 BankAccount SPL

We used parts of the BankAccount SPL throughout the thesis to illustrate aspects in our discussions. In Figure 2.1 on Page 7, we show the feature model of the software product line. The BankAccount SPL contains ten features and 144 possible configurations. For this evaluation, we use and extend this product line, which we obtained from Thüm et al. [2014c]. We provide all extended versions of the product line at the software product line example repository of FeatureIDE. We use this software product line because it was already used in similar previous work [Bubel et al., 2014; Meinicke, 2013; Praast, 2014; Thüm et al., 2014c, 2012a] and it is a small software product line that already contains most aspects we discussed in Chapter 3 regarding feature stub and metaproduct generation.

As we focus on the reuse potential of abstract contracts and feature-family-based verification, we verify the case study in several versions. The first version represents our base version. As we wanted to include `assingable` clauses into our experiments, we added them to the version one. For versions two, three, and four, we examined the BankAccount SPL for potential improvements of code and contracts. Version five

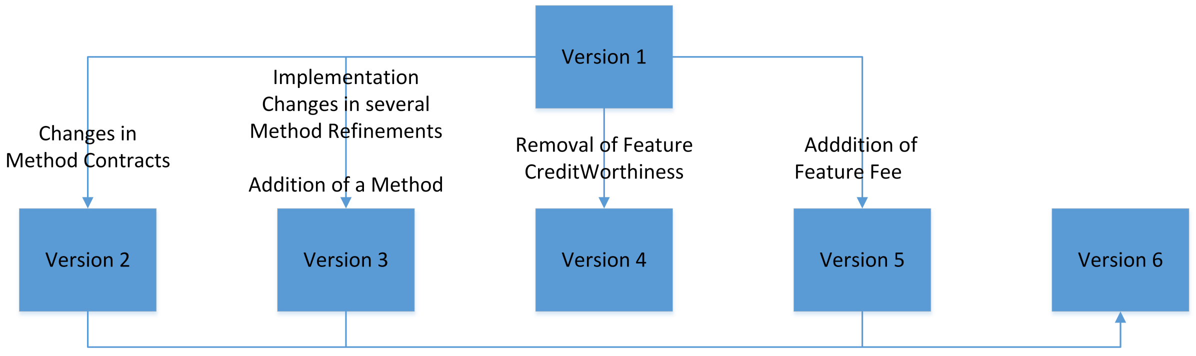

Version 1
Implementation Changes in several Method Refinements
Addition of a Method
Changes in Method Contracts
Removal of Feature CreditWorthiness
Adddition of Feature Fee
Version 2
Version 3
Version 4
Version 5
Version 6

Figure 5.1: Overview of Evaluated Versions

was inspired by Bubel et al. [2014]. Version six was designed to include the maximum amount of changes in one version. In Figure 5.1, we show a schematic representation of the versions and how they are related to each other. We describe the first one in more detail and give a short summary for the subsequent versions. We evaluate all other versions as direct successors of version 1, regardless how big the changes are. We designed all versions, so that the verification is successful (i.e., all proof goals can be closed).

Version 1

The first version is the complete BankAccount SPL with all features. To give the reader a better understanding of the functionality, we give a brief overview about the software product line. As we already introduced the features *BankAccount*, *DailyLimit*, *CreditWorthiness*, and *Transaction* in the previous chapters, we focus on the remaining features. In the base feature *BankAccount*, the possible overdraft limit is set to `0`. The feature *Overdraft* refines this field and actually sets the value to -5000. Feature *Interest* introduces the calculation of interest for the account by adding method `calculateInterest`. This functionality is extended by feature *InterestEstimation*, which makes it possible to predict an account's interest until the end of the year. Feature *Lock* contains a locking functionality for accounts, so that the account cannot be changed by another thread. This functionality is employed by feature *Transaction*, which transforms account updates into atomic transactions. With feature *Logging*, account updates can be logged. Finally, feature *TransactionLog* extends this logging functionality to transactions implemented by feature *Transaction*.

Version 2

For version two, we changed several contracts. First, we enriched method `nextYear` in role *Application* of feature *BankAccount* with a method contract. Furthermore, we removed the **`requires`** clause of method `credit` in role *Account* of feature *CreditWorthiness*. Finally, we changed the **`ensures`** clause of method `calculateInterest` so that it is more precise. We show both **`ensures`** clauses in Listing 5.1.

```
/*version one:*/
ensures (balance >=0 ==>\result >= 0) && (balance <=0 ==>\result <=0);
/*version two:*/
ensures \result <==> (balance * INTEREST_RATE/36500);
```

Listing 5.1: *Ensures* Clause of Method *calculateInterest* in Versions One and Two

Version 3

In version three, we add a new method `overdraftLimitExceeded` to role *Account*. The method checks whether a given value is smaller than the allowed overdraft of the account. We show both its implementation and contract in Listing 5.2. Additionally, we change the implementation of different method refinements of `update` and `undoUpdate` in role *Account* without needing to change their respective contracts. In role *Account* of feature *BankAccount*, which we show in Listing 2.2 on Page 9, we changed the assignment of the field `balance`. While the value is newly calculated in version one, we now set it to `newBalance`, as can be seen in Line 18 of Listing 5.2. Additionally, we replace the original if-statement by a call to our newly introduced method `overdraftLimitExceeded` (see Line 16 in Listing 5.2). In role `Account` of feature *DailyLimit*, we remove a unnecessary if-statement from method `undoupdate`. Finally, we reverse the if-statement in both `update` and `undoUpdate` method in role *Account* of feature *Logging*. We show both versions of `update` in Listing 5.3.

```
 1   /*@ ensures \result <==> (newBalance < OVERDRAFT_LIMIT);
 2     @ assignable \nothing;
 3     @*/
 4     private boolean overdraftLimitExceeded(int newBalance) {
 5         return newBalance < OVERDRAFT_LIMIT;
 6     }
 7
 8     /*@
 9      @ requires x != 0;
10      @ ensures (!\result ==> balance == \old(balance))
11      @   && (\result ==> balance == \old(balance) + x);
12      @ assignable balance;
13      @*/
14     boolean update(int x) {
15         int newBalance = balance + x;
16         if (overdraftLimitExceeded(newBalance))
17             return false;
18         balance = newBalance;
19         return true;
20     }
```

Listing 5.2: Role *Account* of Feature *BankAccount* in Version Two

```
    /*original version:*/
    boolean update(int x){
        if (original(x)){
            updateCounter = (updateCounter + 1) % 10;
            updates[updateCounter] = x;
            return true;
        }
        return false;
    }
    /*version three: */
    boolean update(int x){
        if (!original(x)){
            return false;
        }
        updateCounter = (updateCounter + 1) % 10;
        updates[updateCounter] = x;
        return true;
    }
```

Listing 5.3: Method *update* in Versions One and Three

Version 4

For version 4, we only remove feature *CreditWorthiness* from the software product line of version one. Thereby, we not only remove the feature from the feature model, but also its source code. This includes the introduction of method `credit` in role *Account* as well as the refinement of constructor of role *Main*.

Version 5

In version five, we add the feature `Fee` to the BankAccount SPL. We show the new feature model in Figure 5.2. The feature implements a fee for every transaction that is successfully performed. The fee only needs to be paid by the source of the transaction. To realize the fee, role *Transaction* is refined. We show the refined role in Listing 5.4. Method `transfer` is only changed slightly. First, instead of subtracting the given amount from the source, the fee is also subtracted. Additionally, we updated the respective **`ensures`** clause (see Listing 5.4). Feature *Fee* is arranged between features *Transaction* and *Transactionlog* because its refinement method `transfer` replaces feature *Transaction*'s method `method` but can still be extended by *TransactionLog*'s functionality.

```
public class Transaction {
    private int FEE = − 105;

    /*@ requires \original;
      @ ensures \result ==> (\old(destination.balance) +
      @ amount == destination.balance);
      @ ensures \result ==> (\old(source.balance) −
      @ (amount + amount*FEE/100) == source.balance);
      @ ensures !\result ==> (\old(destination.balance)
      @ == destination.balance);
      @ assignable source.lock, destination.lock, source.balance,
      @ destination.balance, source.withdraw, destination.withdraw;
```

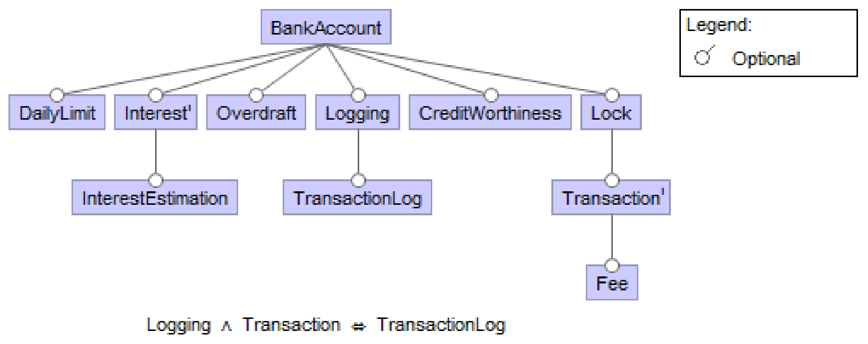


Figure 5.2: Feature Model for BankAccount SPL with Feature Fee

```
        @*/
      public boolean transfer(Account source, Account destination,
       int amount) {
          if (!lock(source, destination)) return false;
          try {
              if (amount <= 0) {
                  return false;
              }
              int sum = (amount*FEE)/100;
              if (!source.update(sum)) {
                  return false;
              }
              if (!destination.update(amount)) {
                  source.undoUpdate(sum);
                  return false;
              }
              return true;
          } finally {
              source.unLock();
              destination.unLock();
          }
      }
}
```

Listing 5.4: Role *Transaction* of Feature *Fee*

Version 6

For version six, we combine three previous versions. In particular, we add feature *Fee*, as we did for version 5. Additionally, we perform the contract changes from version 2 and, finally, we change the several method implementation from version 3. By that, we combine three different kinds of changes and achieve an overall bigger software product line.

5.2 Experimental Design

We evaluate five different approaches. For each of the approaches, we verify all aforementioned versions to test them individually with regards to their efficiency during code evolution. We aim to compare the family-based verification developed by Thüm et al. [2012a] and our concept proposed in this thesis. Although it is common practice to compare new verification approaches to unoptimized product-based approaches, we do not include these approaches into this evaluation because it is already established that these approaches are inferior to family-based approaches[Thüm et al., 2014a]. Hence, we are also the first to compare family- and feature-family-based approaches with each other. Furthermore, our approach also includes a family-based phase, so it is reasonable to compare it with other family-based approaches. However, there are several steps in between these two approaches and only performing a verification with both these strategies does not do justice to them. We show possible differentiating parameters between the approaches in Table 5.1.

Table 5.1: Comparison Parameters For Family-Based Verification Strategies

Parameters	Values
Contracts	Concrete - Partially Abstract - Abstract
Partial Proofs	No - Yes
Metaproduct	Thüm et al. [2012a] - Apel et al. [2013e]
Verification	Inlining - Use Contracts
Proof Replay	No - Reuse of Partial Proofs - Reuse of Full Proofs

Subsequently, we describe for both the family-based verification approach by Thüm et al. [2012a] and our approach, which values for the attributes in Table 5.1 they represent. Thüm et al. [2012a] used concrete contracts and designed their own metaproduct that can be proven by a regular theorem prover with Method Inlining. However, they neither compose proofs, nor do they plan to reuse the obtained proofs in any way. We use abstract contracts in both the feature-based and the family-based phase to realize *feature module dependencies*. In the feature-based verification, we obtain *partial proofs* that we can use to compose proofs in the family-based approach. We can also reuse the *partial proofs* after code evolution. Finally, we do not inline called methods but instead only use their contracts.

There are apparent differences between these two approaches that can lead to either benefits or drawbacks. To make a fairer comparison and to measure the influences of the parameters described above, we evaluate not only the two discussed approaches but also three *intermediate strategies*. We are aware that these three approaches do not cover all possible influences but a detailed analysis and evaluation of each possible combination of attributes is beyond the scope of this thesis. In Figure 5.3, we provide a schematic diagram to illustrate the differences.

The first approach represents the verification strategy developed by Thüm et al. [2012a] and Meinicke [2013]. We generate the metaproduct for each version and verify them

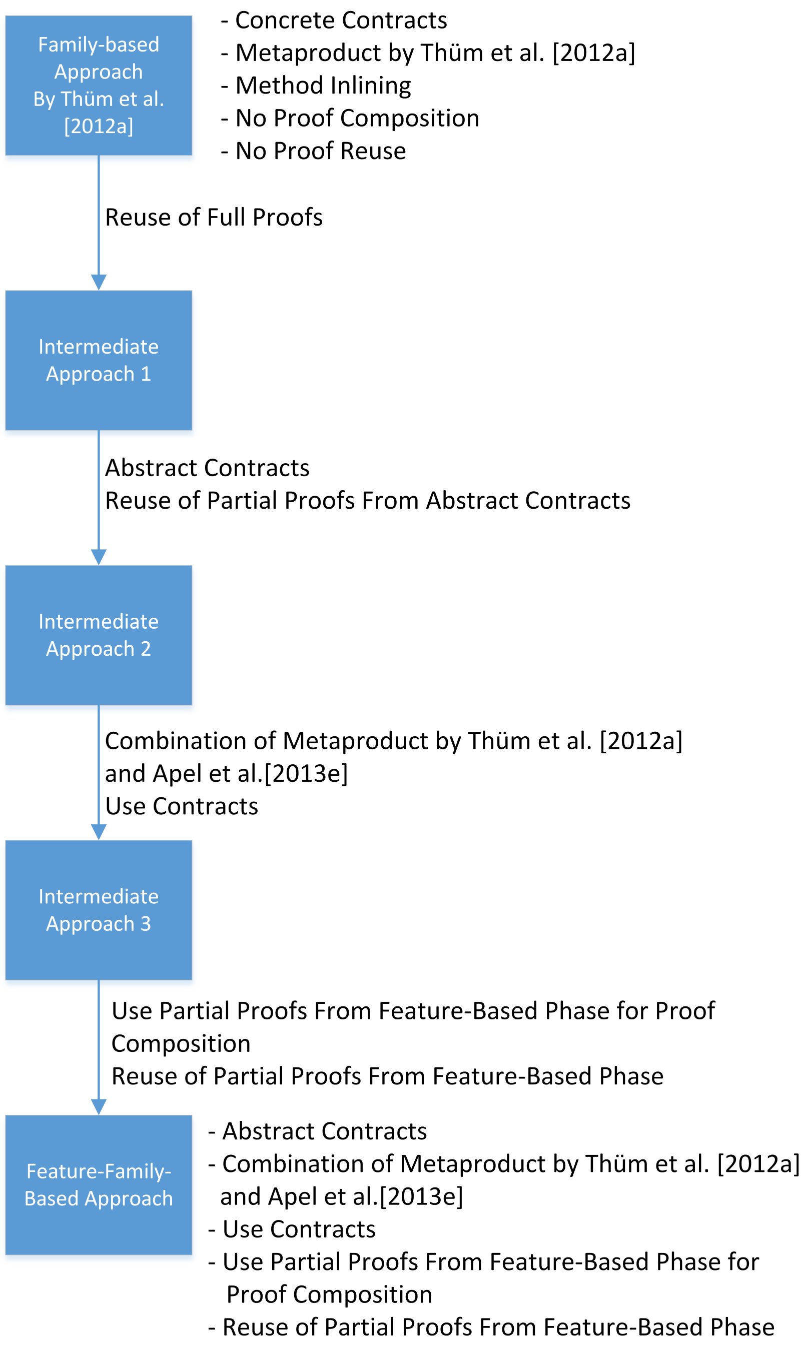

Family-based Approach By Thüm et al. [2012a]
- Concrete Contracts
- Metaproduct by Thüm et al. [2012a]
- Method Inlining
- No Proof Composition
- No Proof Reuse
Reuse of Full Proofs
Intermediate Approach 1
Abstract Contracts
Reuse of Partial Proofs From Abstract Contracts
Intermediate Approach 2
Combination of Metaproduct by Thüm et al. [2012a] and Apel et al.[2013e]
Use Contracts
Intermediate Approach 3
Use Partial Proofs From Feature-Based Phase for Proof Composition
Reuse of Partial Proofs From Feature-Based Phase
Feature-Family-Based Approach
- Abstract Contracts
- Combination of Metaproduct by Thüm et al. [2012a] and Apel et al.[2013e]
- Use Contracts
- Use Partial Proofs From Feature-Based Phase for Proof Composition
- Reuse of Partial Proofs From Feature-Based Phase

Figure 5.3: Overview of Evaluated Approaches

in isolation with Method Inlining. We chose this approach because it is the current approach to family-based verification as the verification of evolving software product lines has not yet been discussed in detail.

Our first *intermediate approach* is to not verify all metaproducts from the beginning but only for version one and save the obtained proofs. We then replay the proofs on the subsequent metaproducts. When a proof cannot be replayed or (as in version three, five, and six) there are methods without any proof, we verify the method from the beginning. We use this approach to determine benefits from reusing proofs.

For our second *intermediate approach*, we again generate the metaproduct of all versions. However, we then transform all method contracts by abstract method contracts. We perform the macro ”Finish abstract proof part” in KeY for each method in the metaproduct of version one and save the result as a *partial proof*. We then reuse the *partial proofs* for all subsequent metaproducts. By using abstract contracts on the metaproduct by citetThum.2012, we aim to obtain information about how abstract contracts help with increasing reuse potential of *partial proofs*.

The third *intermediate approach* represents our family-based phase from Section 3.2. Similarly to the approach directly above, we perform the macro ”Finish abstract proof part” in Key for each metaproduct of version one and then reuse them for the metaproduct of the following versions. However, instead of Method Inlining (see Section 2.3), we do use the contracts of called methods. The use of contracts requires us to make several adjustments to the BankAccount SPL to be able to prove it because callers do not get to know the method's implementation but only rely on the contract. Therefore, we add several invariants and make method contracts more specific. We explain these changes in more detail in Appendix B, as we do not consider this information necessary to the understanding of this evaluation but nonetheless want to document our work. We evaluate this approach to examine the differences between inlining of called methods or using their contracts during verification.

Our last approach represents our concept from Chapter 3. We first create a feature stub for all features and perform a feature-based verification on each of them. For the family-based verification phase, we generate the same metaproducts as in the third *intermediate approach* but then reuse the obtained *partial proofs* from the feature-based verification. We include this approach in our evaluation to determine whether our approach of combining abstract contracts and feature-based verification really is superior to the solely family-based approaches with regards to product-line evolution.

The evaluation is performed by means of our existing tool support. We use FeatureIDE to generate the metaproducts for versions one through three and feature stubs, and employ KeY for verification. As pointed out in Chapter 4, we neither implemented the check for changes in feature stubs before every feature-based verification nor the metaproduct generation for our feature-family-based approach. We simulate the check by re-creating all feature stubs and checking for changes manually. Afterwards, we only re-verify changed feature stubs. We also only build the metaproduct for versions four and five by hand.

We compare these approaches regarding different aspects. First, we examine the overall verification effort of all approaches. Therefore, we analyze the proof complexity in terms of necessary proof steps for a verification and the verification time. As verification time, we use the sum of all method verification times. To reduce measurement errors, we perform each verification for three times and take the mean value of the three. Second,we present the results from the perspective of product line evolution and determine the actual potential of proof reuse for each approach. Therefore, we take the verification of version one as baseline and examine the savings for the verification of the following versions. We evaluate both overall verification effort and reuse potential individually because, even with a high rate of reusability, significantly larger and more complex proofs diminish the effect of proof reuse. Likewise, if the proofs' complexity is rather small, even the reuse of fewer steps is helpful. Third, we discuss the usability of each of the approaches. We are aware, that this metric is rather subjective. Nonetheless, we find it important because reusing of proofs aims to help simplifying the verification of programs. If reusing, however, is more complicated for the user while the saved computational benefits are rather small, the approach might not be helpful after all. For the evaluation, we used a notebook with Intel Core i7-3610QM CPU @ 2.30GHz with 8 GB RAM on Windows 7 and Java 1.7.

There are several other possible metrics, we do not include in our evaluation. Subsequently, we discuss them shortly and explain why we do not consider them. For the generation of the feature stub, we perform a type check with Fuji. We could also measure the type check time as only the last of the above discussed approaches includes this type check which can be seen as a time disadvantage. Similar aspects can be discussed about the time for loading of the projects in KeY, the adaption of the *partial proofs*, replaying the *partial proofs*, feature stub, or metaproduct generation. We do not consider these times as we have not implemented all of them so they could be performed automatically. The measured times would largely depend on how fast we performed the manual steps and, thus, cannot be fairly compared. On the other hand, by not considering these times, we skew the measured verification time in favour of the approach we developed over the course of this thesis.

5.3 Results

To present our results, we present them in two parts. First, we compare the proof complexity and verification times of the different approaches. In the second part, we show our results regarding reuse potential of all approaches.

5.3.1 Verification Effort

In Section 5.3.1, we show the results of our evaluation. Each column contains the number of proof steps necessary to prove a version of the BankAccount SPL for an approach. As we only look the approaches themselves, we include all proof steps including from reused *partial proofs*. For approaches intermediate two and three, we separate the data in proof steps for abstract contracts and proof steps for concrete contracts. For

Table 5.2: Proof Steps for Verification

	Family-Based	Intermediate 1	Intermediate 2		Intermediate 3		Feature-Family-Based	
	Concrete	Concrete	Abs	Concrete	Abs	Concrete	Feature-based	Family-Based
V1	200556	200556	17637	211913	16110	102380	7715	71175
V2	211795	185371	17568	192362	16110	83154	7480	66433
V3	218527	203629	17681	196235	15690	88695	6407	66526
V4	220042	180195	17539	198873	13589	100399	7438	65364
V5	338544	338544	17583	322360[a]	12559	95826	7480	86552
V6	341931	341931	17734	332449[a]	12185	108319	6401	98655
Avg	255232,5	241704,3	259989[a]		110836		82937.7	

[a]A complete verification was not accomplished.

the feature-family-based approach, we separate the data into the both phases. In the last row, we show the respective averages of needed proof steps. We also present the results as a histogram in Figure 5.4 to simplify the comparison between the different approaches. For versions five and six of intermediate approach two, we were not able to close all proof goals for method `transfer` and stopped KeY at a little over 300.000 proof steps. As these results are not representative, we do not include them into the histogram.

Both Section 5.3.1 and Figure 5.4 show that the first *intermediate approach* yields only small advantages compared to the family-based approach by Thüm et al. [2012a]. On average intermediate approach 1 needs 241700 proofs steps, while the family-based approach needs 255200, so that only about 6% are saved.

Our second *immediate approach* provides similar results. However, the comparability is limited because of the unclosed proof goals that we mentioned above. The verification for version one needs slightly more computational effort due to the reasoning about abstract contracts. For version two, three and four, we find very small advantages compared to the family-based approach but disadvantages compared to the first *intermediate approach*. As we do not have full verification of all versions for the first `intermediate approach`, we cannot fully compare this approach to the others.

Our third *intermediate approach* yields enormous gains regarding proof steps. On average, the approach needs 11,0800 proof steps to accomplish the verification, which is about 57% fewer proof steps than the family-based and 54% fewer proof steps than *intermediate approach* one needed. We were, however, not able to prove all methods with the option "Use contracts" in KeY. We could verify method `transfer` by means of "use contracts" but needed to use Method Inlining for the other methods in its calling hierarchy (see Figure 3.1). For versions one to four this includes `transfer_Transaction` and `transfer_TransactionLog` For versions five and six, we needed to prove `transfer_TransactionLog` and `dispatch_transfer_Fee`.

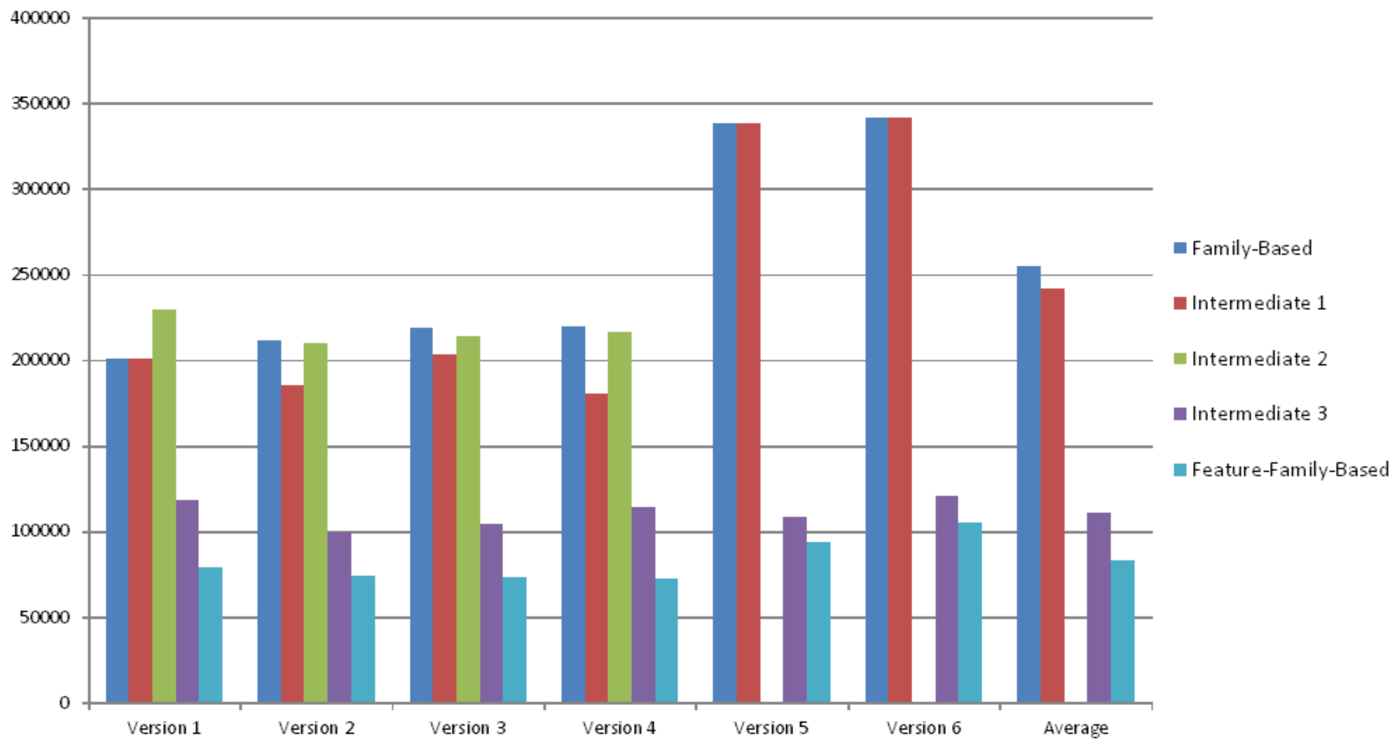


Figure 5.4: Comparative Overview of Proof Steps

Our approach is yet another improvement to the third *intermediate approach*. Only 82900 proof steps are necessary on average to verify each version. This means a decrease of 68% compared to the family-based approach and 25% compared to *intermediate approach* three.

In Table 5.3, we show the times needed for the respective verification. Each column contains the time in milliseconds necessary to prove all methods. We give average times for each approach in the last row. In Figure 5.5, we present the results as a histogram for an easier comparison of the different approaches.

The results shown in Table 5.3 and Figure 5.5 resemble the results discussed above, so that we only discuss them briefly. Again, the family-based approach and the first *intermediate approach* provide similar results. *Intermediate approach* three and our feature-family-based approach are about 50% faster compared to the family-based approach (41.5s vs. 21s). Again, we do not include the second *intermediate approach* into the histogram because of incomplete proofs.

5.3.2 Product-Line Evolution

In Table 5.4, we present the savings due to proof reuse for the. For each approach, we show the proof steps that can be provided by *partial proofs* both as an absolute and an relative value. The last row of the table contains the average savings for each approach. Again, we leave out the results for versions five and six of *intermediate approach* two

Table 5.3: Comparative Overview of Verification Times

Verification Time in ms	Family-Based	Intermediate 1	Intermediate 2	Intermediate 3	Feature-Family-Based
V1	308476	307650	289851	205468	212896
V2	313699	294420	254721	186359	161767
V3	319915	312805	309607	190885	182356
V4	317346	266439	343470	226885	157701
V5	627794	640230	1009529[a]	260192	222586
V6	599297	614169	1012394[a]	256845	231327
Avg	414421.2	418834.3	556010.5[a]	232621.7	194396.7

[a]A complete verification was not accomplished.

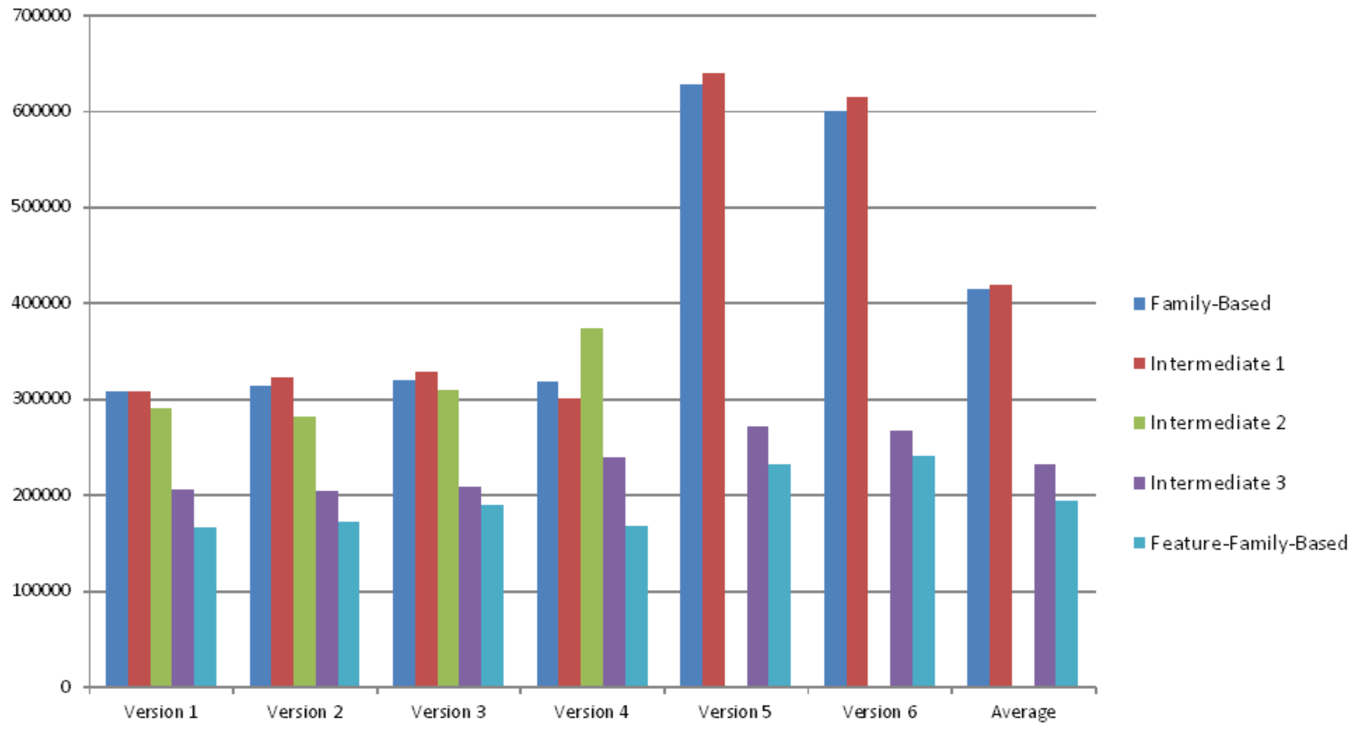


Figure 5.5: Comparative Overview of Verification Times

Table 5.4: Comparative Overview of Reused Proof Steps

	Family-Based		Intermediate 1		Intermediate 2		Intermediate 3		Feature-Family-Based	
Version 1	200556		200556		229550		118490		78890	
	Replayed Proof Steps		Replayed Proof Steps		Replayed Proof Steps		Replayed Proof Steps		Replayed Proof Steps	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Version 2	0	0	29137	14.5	17540	7.6	16110	13.6	7480	9.5
Version 3	0	0	14897	7.4	11028	4.8	15108	12.8	5601	7.1
Version 4	0	0	29255	14.6	17539	7.6	11062	9.3	7438	9.4
Version 5	0	0	0	0	17583[a]	7.6	10616	9	7480	9.5
Version 6	0	0	0	0	11043[a]	4.8	9608	8.1	5585	7.1
Average	0	0	14657.8	7.3	14946.6[a]	6.5	12500.8	10.6	6716.8	8.5

[a]A complete verification was not accomplished.

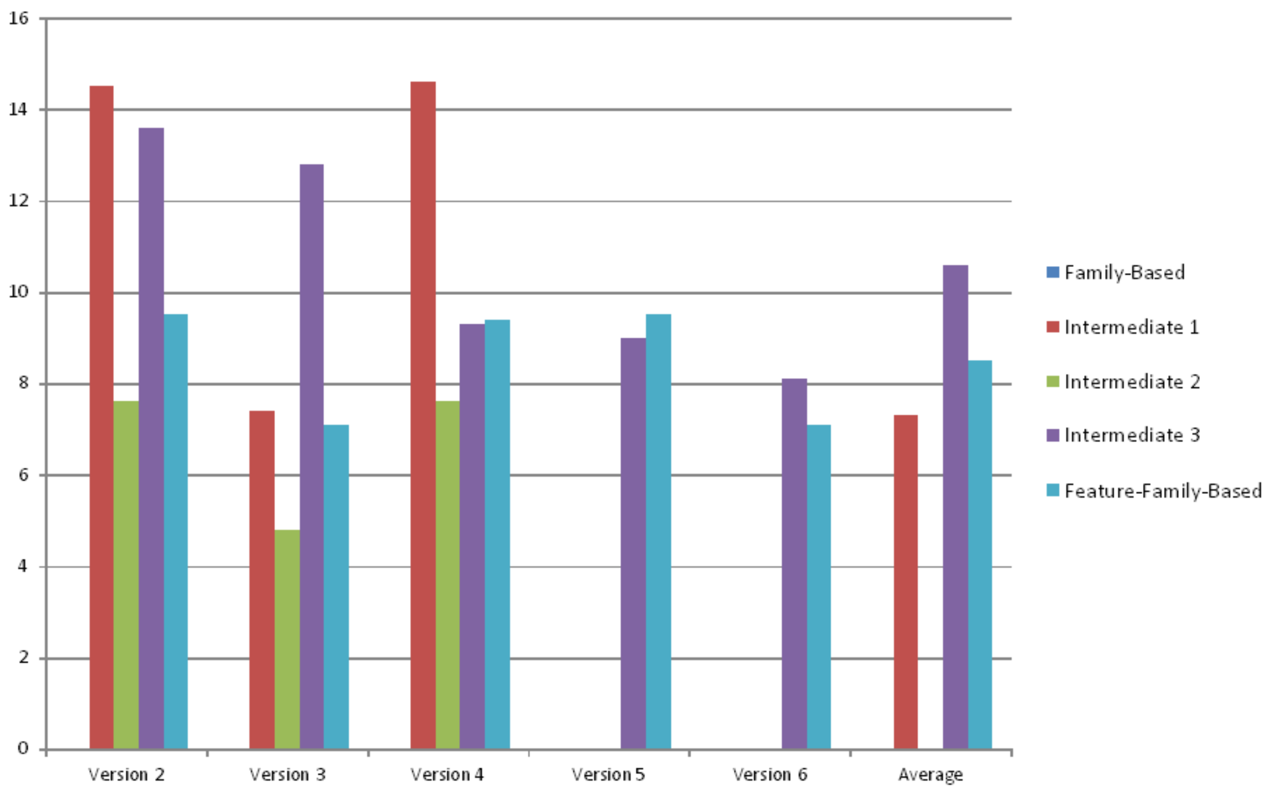


Figure 5.6: Comparative Overview of Reused Proof Steps

because of incomplete proofs. We also provide the relative values as a histogram in Figure 5.6.

In the following, we compare the relative values of reused proof steps because if the original proof is significantly higher, the informative value of higher absolute values of reused proof steps is rather limited. For the family-based approach, no proofs are reused, because we defined that scenario, so that we generate the metaproduct each time. However, for the first *intermediate approach*, we were able to reuse proofs for the verification of versions one to three. As we save full proofs, only if both contract and a method's implementation stay unchanged, the saved proofs can be reused. Therefore, we have high rates of reuse for small changes but no reuse for versions with bigger changes.

The second *intermediate approach* provides a more steady reuse potential. For the three versions, we have complete proofs, we can reuse between 5 and 7.5% of a previous proof. In this approach, the *partial proofs* contain reasoning about abstract contracts- Therefore, a proof may be even reusable if a method's contract changes. However, as we do not have the data for all versions, a general comparison is hard to perform.

With the third *intermediate approach*, we achieve the highest relative reuse values. On average, 10.6% can be reused. Again, we see that the bigger the changes are, the less we can actually reuse.

Finally, for our feature-family-based approach, we achieve reuse rates of about 7 to 9.5% and 8.6% on average. This means a decrease by two percentage points compared to the third *intermediate approach* but is nonetheless better than *intermediate approaches* one and two. We suspect the decrease compared to *intermediate approach* three to be caused by the complete reverification of a feature stub. We noted during the evaluation, that when a *partial proof* of one feature's method is invalidated, the *partial proofs* of the feature stub may still be valid. By reverifying the whole feature stub, we still perform redundant verification.

5.4 Discussion

A starting point of our work was, that product line evolution is not discussed enough in current approaches to theorem proving of software product lines. Therefore, we designed our approach with the focus of increasing reuse potential. Our goal was to replace proof finding processes with proof checking processes as checking an existing proof is easier and faster than finding a proof to reduce the overall effort. Considering the results, our approach achieves these goals. First, we could enormously reduce the proof complexity and verification time compared to existing family-based approaches. Second, we also accomplish reuse potential of about 8.6%.

The greatest benefits regarding necessary proof steps are achieved in the third *intermediate approach*. The differences between this approach and the second *intermediate* are the use of contract instead of Method Inling during verification and the use of the metaproduct discussed in Section 3.2. The structure of the two metaproducts are relatively similar. Therefore, we suspect the benefits mostly resulting from using the contracts of called methods instead of inlining them. Therefore, a called method is not symbolically executed every time it is called, which leads to significantly less complex and therefore smaller proofs. Additionally, as mentioned in Section 5.2, we made several method contracts more specific and created invariants to limit the frame that KeY has to consider. These changes also cause the proofs to be smaller.

However, these changes are complex to perform and cause the developer high effort. We needed to run KeY several times and analyze the failed verifications until we got the specification to be precise enough for a successful verification. For existing projects, these changes limit the time gain for a developer who is unexperienced in theorem proving. However, if they develop a software product line and its specification with this in mind, it might still be worth it.

We want to address the heavy influence of method `transfer` and its refinements regarding both needed time and necessary proof steps. Although the BankAccount consisted of up to 18 methods, depending on the used version, method `transfer` is responsible for most of time and steps due to its multiple refinements and many *feature module dependencies*. The better an approach could deal with this method, the better its results were. By adding feature *Fee* for versions five and six, we yet increase the influence of method *transfer*.

So far, we mostly discussed the issue of necessary proof steps and verification time. In Section 5.2, we mentioned a third aspect, which we want to discuss now. Theorem proving requires expertise and effort to be performed. This holds for all evaluated approaches. However, there are significant differences regarding effort for the developer. First, we had to deal with saved *partial proofs*. They need to be saved at a location, from which they can later be reused. For our evaluation, this was relatively easy, as we performed all verifications in a row. In practice, however, the time spans between verifications are probably bigger. The same holds for feature stubs. To be able to identify changes since the last feature-based verification, the feature stubs the verification was performed on need to be stored, too. We also needed to perform the adaptions for the family-based by hand. Second, we had to build all the metaproduct, we proposed in this thesis, by hand. This process was complicated due to the several versions, we had to create and our goal to perform the verification by means of Use Contracts. Finally, there were complications during the verification for method `transfer`, which caused additional effort and re-verification. Most of these aspects can be automated by tool support, which should lead to an easier verification. However, the storage of *partial proofs* and feature stubs remain a task for the developer.

5.5 Summary

In this chapter, we performed five different verification approaches on six different versions of the BankAccount SPL. Our developed feature-family-based approach was successfully applied to the software product line. We found that our approach reduces the amount of proof steps by more than 65% compared with the family-based approach by Thüm et al. [2012a]. We were also able to reuse about 8.5% of the original proof. These gains come with enormous overhead regarding storing and handling of feature stubs and *partial proofs*, most of which can be reduced by additional tool support. Furthermore, the third *intermediate approach*, which requires less overhead, provides better outcomes regarding proof reuse (about 10.5%), but results in about 25% more complex proofs.

6. Related Work

This thesis is mainly concerned with software product lines and their verification. We thereby employed concepts like abstract contracts and feature stubs and touched topics like change impact analysis in the context of software product line. In this chapter, we present related work on these topics.

Analysis of Software Product Lines

Thüm et al. [2014a] conducted a survey about software product line analysis techniques such as model checking, type checking and theorem proving. They classify them into product-, family-, feature-based and combinations thereof and analyzed benefits and weaknesses of the different strategies. Based on their work, we classify our verification approach as feature-family-based.

A wide range of analyses have been applied to software product lines. These include analyses genuinely originating within the paradigm software product lines including feature-model analysis [Bubel et al., 2010; Jörges et al., 2012; Sabouri and Khosravi, 2010]. However, there also are techniques known from regular programs and only transformed to software product lines such as syntax checking [Kästner et al., 2011], type-checking [Apel et al., 2010a, 2008; Bettini et al., 2014; Kästner et al., 2012], model checking [Apel et al., 2011; Classen et al., 2010; Fantechi and Gnesi, 2008], and theorem proving. As we develop a concept for theorem proving of software product line, we examine the research to theorem proving of software product lines in more detail in the following part.

Theorem Proving of Software Product Lines

Harhurin and Hartmann [2008] propose a product-based approach to software product line verification in which all possible products are generated and verified. This unoptimized approach, however, is problematic, because the amount of products increases up to exponentially with every added feature. To optimize the verification, Bruns et al. [2011] propose a different verification approach for delta-oriented software product lines. At first, only one base product is verified. Then, all other variants are created upon this base product, which may lead to invalidation of parts of the proof. Only the invalidated proofs need to be proven again.

Nonetheless, the product-based strategies become infeasible for huge software product lines. Thüm et al. [2012a] propose a family-based approach for feature-oriented programming, which uses variability encoding. They create a single metaproduct incorporating a software product line's variability, so that it can simulate the whole software product line. A metaspecification from the software product line is created. They are able to reduce the verification time of a previous version of the BankAccount SPL compared to an unoptimized product-based approach by 85%. Meinicke [2013] implements and extends the metaproduct, proposed by Thüm et al. [2012a]. Apel et al. [2013e] compared different strategies and proposed a slightly different approach to variability encoding. They also find that familiy-based approaches are superior to other strategies. For our concept, we adopt parts of both metaproducts. In our evaluation, we discuss that the family-based approach by Thüm et al. [2012a] to software product line theorem proving is simpler to realize. However, they do not consider product line evolution in their metaproduct design. Due to the structure of our metaproduct, we can easily replay the partial proofs, generated in the feature-based phase.

Some approaches combine feature-based with either family-based or product-based strategies. Thüm et al. [2011] propose an approach to compose proofs from smaller proofs. They create partial proofs for each feature and composed them to generate a proof for each variant. Damiani et al. [2012] take a similar approach for delta-oriented programming. They first prove each implementation unit in isolation. They generate all products, apply the partial proofs and, finally, prove the open proof goals for each product. Other approaches that work similar are provided by Delaware et al. [2011, 2013] and Gondal et al. [2011]. Our feature-family-based approach works similar in the feature-based verification phase. We also obtain partial proofs from this phase that we compose to full proofs in the family-based phase. Nonetheless, for their product-based phase, they generate all variants and so suffer from the same problems solely product-based approaches do.

There is also research concerned with improving effectiveness and efficiency of theorem proving of software product lines. Thüm et al. [2014c] combine theorem proving and model checking and evaluate their approach for potential synergies. They find that both model checking and theorem proving are more effective and efficient when combined especially for software product lines with many defects. A second approach for improvement is increasing the potential for proof reuse with abstract contracts, even though they had not yet been applied to software product lines until our thesis. We give an overview about this topic's research in the next paragraph.

Abstract Contracts

Hähnle and Schaefer [2012] apply the Liskov principle to delta-oriented programming. They aim to provide an approach for proof reuse as software changes often. However, the approach is rather restrictive. So, Hähnle et al. [2013b] proposed abstract contracts as a mean to increase proof reuse potential. They use abstract contracts to provide placeholders that are independent from the actual contract and can be verified regardless whether implementation or contract change. The concept is further extended by Bubel et al. [2014]. They also extend abstract contracts to abstract invariants and provide tool support. Furthermore, they evaluate their implementation and find that, for code evolution, abstract contracts reduce the number of proof steps. We adopt the concept of abstract contracts for our concept in order to both realize *feature module dependencies* in the feature stubs and increase reuse potential of proofs. Bubel et al. [2014] also examine an intermediate concept of partially abstract contracts, whose **`assignable`** clause is not abstract but concrete. They find partially abstract contracts achieve higher savings regarding proof steps. However, they use abstract contracts for regular programs, while we use them for software product lines.

Other Topics

Thüm et al. [2013] discuss the issue of determining inconsistencies between a feature's behaviour and its specification. To solve this issue, they propose a new feature composition mechanism that includes behavioural feature interfaces. These interfaces can then be used to determine faulty features. In the metaproduct, when the theorem prover cannot close all goals, it is difficult to determine the feature or feature combination causing this problem. Behavioural feature interfaces provide a mechanism for an easier localization of such errors.

A focus of our thesis is code evolution of software product lines. Passos et al. [2013a] discuss relevant aspects of code evolution from a feature-oriented perspective. Among other things, these aspects were consistency checking, change impact analysis and considerations regarding the general architecture of a software product line. In our work, we need change impact analysis for re-verification checks of feature stubs. When a software product line evolves after performing our feature-based verification, a previous proof of a feature might be invalidated. A change impact analysis on the respective feature stub can help determine whether a re-verification is necessary or if the proof is still valid.

7. Conclusion

Software product lines represent a mechanism to develop a set of program variants with a common code base. By that, they provide several chances to reduce the overall costs of development. First, they make it possible to reuse existing code instead of developing each variant from the beginning or copying the code from different projects. Software product lines also reduce the effort to fix errors in the code, because an error only needs to be fixed once per software product line and not for each variant individually. This incorporated variability is implemented by features. Each feature represents a behaviour of a software visible to the user of the software. Different features interact with each other to realize a variant's desired functionality. However, due to these interactions, new errors can be introduced.

A software's behaviour can be verified. A verification becomes especially important when software artefacts are used in safety-critical systems. Theorem provers are tools for the verifications of software. If the software's behaviour is specified, they can use this specification to prove a program's correctness with respect to their contracts. For regular programs, theorem proving techniques are advanced, but for software product lines, theorem proving is more complex and difficult. Three main strategies were established as means to verify product lines. Product-based approaches generate all variants of a software product line and verify them individually. In family-based approaches, either the theorem prover is extended to support software product lines or a metaproduct is generated, that incorporates a software product line's variability in both implementation and specification. This metaproduct is then verified. Finally, for feature-based approaches, features are verified in isolation. As these different strategies come with benefits and weaknesses, combinations of them were suggested.

In this work, we developed a verification strategy that combines a feature-based and a family-based approach. Our goal was to create an approach that provides the possibility of reusing created proofs in a previous verification. We combined these two strategies as family-based strategies require the metaproduct to be generated completely each time a verification is performed. Therefore, nothing is considered to be reused. To increase the reuse potential, we performed a feature-based verification. Feature-based verification had not been realized before because features interact with each other and these interactions cannot be realized by a solely feature-based approach. To realize at least intentional feature interactions, we created feature stubs and transformed all concrete contracts into abstract contracts. Abstract contracts contain placeholders that represent the different clauses of a contract and can be used to reason about a method's correctness without a concrete definition. However, if not all placeholders are defined, not all proof goals can be closed. Then we performed the feature-based theorem proving of the feature stubs and then saved the created partial proofs. In the family-based, we created a metaproduct that incorporates the variability of the software product line and can therefore simulate all its products. To be able to able to reapply the proofs obtained in the feature-based verification, we also generate all method contracts as abstract contracts. After replaying the partial proofs on the metaproduct, the proofs can be closed by a theorem prover.

We also provide tool support with this work. Our tool supports the feature-based phase. In particular, it creates feature stubs with the necessary abstract contracts. After the generation, each stub is verified. We implemented our tool as an extension to FeatureIDE and FeatureHouse as well as the theorem prover KeY.

To evaluate our concept, we verified the BankAccount SPL with five different approaches in six different versions. These approaches include a family-based approach, our new feature-family-based approach and three *intermediate approaches* to make an analysis, which of the used concepts cause which effect, easier. We used FeatureIDE to generate the feature stubs and metaproducts and KeY to perform the verification. We were able to reduce the overall amount of proof steps for the verification by 65% and reduce the verification time by 50% compared to the family-based approach. Furthermore, in our approach, we reused around 8.5% of the proofs. However, these gains come with a large management overhead.

Future work should expand tool support for two reasons. First, most of the overhead work can be automated for example the metaproduct generation or the checking process before each feature-based verification, if a feature must be verified again or if its existing proof is still valid. Second, we discussed the limits of our evaluation in Chapter 5 due to the missing tool support. With a full tool support, the different approaches could be compared more fairly regarding their computation time.

In other work, feature-based verification was used to obtain proofs that can be reused for a subsequent product-based verification. However, they were never combined with abstract contracts. Future work should examine whether feature-based approaches combined with abstract contracts provide a good base for a subsequent product-based verification. They are not able to compensate the inherent weaknesses of product-based approaches, but may help reducing the amount of verification.

In our family-based phase, we used a metaproduct that made several compromises in order to provide compatibility with proof reuse. Future work should examine whether a metaproduct in a different structure is able to provide similarly good support for proof reuse. Especially, the handling of constructors and the calling hierarchy should be explored.

For our evaluation, we elaborated on parameters, in which family-based approaches might differ from each other. This discussion should be intensified as it can be helpful to examine the effects of different parameter values on both verification time and proof steps. Especially, the advantages and disadvantages of Method Inlining and Use Contracts for software product line verification should be analyzed. A more advanced analysis can also help with creating not yet known approaches by combining parameter values. The last of our intermediate approaches seems to be a promising candidate for further examination.

We only evaluated our approach with one software product line and in comparison family-based approaches. As mentioned above, there are also feature-product-based approaches and solely product-based approaches. These should also be compared with our approach.

In this work, we only focussed on method contracts. However, the concept of abstract contracts has been expanded to class invariants as well. Therefore, the effects of abstract invariants for product line verification should also be explored. Finally, abstract method contracts focus on the most basic constructs of JML. Other JML keywords may also be used for abstract contracts.

For this thesis, we limited the scope to explicit contract refinement. There are other ways of contract composition (e.g., contract overriding, conjunctive contract refinement, or consecutive contract refinement), though. Hence, future research should extend our metaproduct generation to these contract composition mechanisms as well.

Existing contract composition mechanisms for feature-oriented programming mainly consider pre- and postconditions, but there is no established way of handling **`assignable`** clauses. For practicality reasons, we merge **`assignable`** clauses over different refinements and calls to other features' methods for the metaproduct. Future work should determine how to proceed with **`assignable`** clauses for product derivation for the different mechanisms.

A. Appendix

In this appendix, we present the feature stubs of features *BankAccount* and *Transaction-Log* in their entirety. We show them as code listings and provide additional information on their structure.

Feature Stub of Feature *BankAccount*

The feature stub consists of four classes. As `BankAccount` is the base feature of the software poduct line, it has no `feature module dependencies`. Therefore, the feature stub is very similar to the feature itself. First, the feature stub includes the `FeatureModel` class, which we show in Listing A.1. The class only contains the feature variable `BankAccount`.

```
package FM;
public class FeatureModel {
    public static boolean BankAccount;
}
```

Listing A.1: Class *FeatureModel* For Feature Stub of Feature *DailyLimit*

The second class represents the role `Main` from the feature. We show the class in Listing A.2. The class is not relevant for the feature stub generation, as none of its methods contains any contract. Therefore, it is adopted from the feature module without any changes.

```
public class Main {
    public static void main(String[] args) {
        new Main();
    }

    private Object a;
    private Object b;

    public Main() {
        a = new Application();
        b = new Application();
        getA().account.update(100);
        getB().account.update(200);
        getA().nextDay();
        getB().nextYear();
    }

    private Application getA() {
        return (Application)a;
    }

    private Application getB() {
        return (Application)b;
    }
}
```

Listing A.2: Class *Main* For Feature Stub of Feature *BankAccount*

The third class represents the role `Account` from the feature. We show the class in Listing A.3. The class only contains methods and fields that were already part of the respective role in the feature, but the contracts of methods `update` and `undoUpdate` as well as the constructor are transformed into abstract contracts.

```
public class Account {
    public final int OVERDRAFT_LIMIT = 0;

    /*@ public invariant this.balance >= OVERDRAFT_LIMIT; */
    public int balance = 0;

    /*@
     @ requires_abs AccountR;
     @ def AccountR = FM.FeatureModel.BankAccount;
     @ ensures_abs AccountE;
     @ def AccountE = balance == 0;
     @ assignable_abs AccountA;
     @ def AccountA = \nothing;
     @*/
    Account() {
    }

    /*@ requires_abs update_BankAccountR;
     @ def update_BankAccountR = x != 0
     @ && FM.FeatureModel.BankAccount;
     @ ensures_abs update_BankAccountE;
     @ def update_BankAccountE =
     @ (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) + x);
     @ assignable_abs update_BankAccountA;
     @ def update_BankAccountA = balance;
     @*/
    boolean update(int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = balance + x;
        return true;
    }

    /*@ requires_abs undoUpdate_BankAccountR;
     @ def undoUpdate_BankAccountR =  FM.FeatureModel.BankAccount;
     @ ensures_abs undoUpdate_BankAccountE;
     @ def undoUpdate_BankAccountE =
     @ (!\result ==> balance == \old(balance))
     @   && (\result ==> balance == \old(balance) − x);
     @ assignable_abs undoUpdate_BankAccountA;
     @ def undoUpdate_BankAccountA = balance;
     @*/
    boolean undoUpdate(int x) {
        int newBalance = balance − x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = newBalance;
        return true;
    }

}
```

Listing A.3: Class *Account* For Feature Stub of Feature *BankAccount*

The final class of the feature stub represents the role `Application` from the feature. We show the class in Listing A.3. The class only contains methods and fields that were already part of the respective role in the feature, but the contracts of methods `update` and `undoUpdate`, as well as the constructor are transformed into abstract contracts.

```
public class Application {
    /*@ public invariant account != null */
    public Account account = new Account();

    /*@  requires_abs nextDay_BankAccountR;
    @ def nextDay_BankAccountR = FM.FeatureModel.BankAccount;
    @ ensures_abs nextDay_BankAccountE;
    @ assignable_abs nextDay_BankAccountA;
    @*/
    void nextDay() {
    }

    /*@ public normal_behavior
    @ requires_abs nextYear_BankAccountR;
    @ def nextYear_BankAccountR = true;
    @ ensures_abs nextYear_BankAccountE;
    @ assignable_abs nextYear_BankAccountA;
    @ def nextYear_BankAccountA = \nothing;
    @*/
    void nextYear() {
    }

}
```

Listing A.4: Class *Application* For Feature Stub of Feature *BankAccount*

Feature Stub of Feature *TransactionLog*

The feature stub of feature *TransactionLog* consists of five classes. Feature *TransactionLog* has `feature module dependencies` to at least two features. Therefore, the feature stub contains some more method and field prototype than the feature stub for feature *BankAccount*. It also includes the `FeatureModel` class, which we show in Listing A.5. The class only contains the feature variable `TransactionLog`.

```
package FM;
public class FeatureModel {
    public static boolean TransactionLog;
}
```

Listing A.5: Class *FeatureModel* For Feature Stub of Feature *TransactionLog*

The first class of the feature stub, which we show in Listing A.6, represents the role *Transaction* from the feature. The role originally only consists of method `transfer`. This method is a refinement of method `transfer` of feature *Transaction*. The method includes a call to a previous implementation of itself by means of the keyword **`original`**. For the feature stub, we generate a method prototype `transfer_original_TransactionLog` and the keyword is replaced by a call to that method prototype. The contract of `transfer` includes the keyword **`original`**, which is used in explicit contract refinement to include contracts from previous refinements. As we ignore other features for the feature stub generation, we do not only transform the method's concrete contract but also remove the concrete section of the contract. Similar can be said for the contract of the method prototype `transfer_original_TransactionLog`. We do not know, which method is the most previous refinement or its contract. Therefore, we enrich the method only with the abstract section of an abstract contract.

```
public class Transaction {
    /*@ public invariant transactions.length == 10; */
    /*@ public invariant transactionCounter > 0 &&
      @ transactionCounter < 10; */
    public LogEntry[] transactions = new LogEntry[10];
    int transactionCounter = 0;

    /*@ requires_abs   transfer_TransactionLogR;
    @ ensures_abs    transfer_TransactionLogE;
    @ assignable_abs transfer_TransactionLogA;
    @*/
    boolean transfer(Account source, Account destination, int amount) {
        if (transfer_original_TransactionLog(source, destination,
        amount)) {
            transactionCounter = (transactionCounter + 1) % 10;
            transactions[transactionCounter] = new LogEntry(source,
                    destination, amount);
            return true;
        }
        return false;
    }

    /*method prototype*/
    /*@ requires_abs   transfer_original_TransactionLogR;
    @ ensures_abs    transfer_original_TransactionLogE;
    @ assignable_abs transfer_original_TransactionLogA;
    @*/
 boolean transfer_original_TransactionLog(Account source,
        Account destination, int amount){}
}
```

Listing A.6: Class *Application* For Feature Stub of Feature *TransactionLog*

The third class is `LogEntry`, which we show in Listing A.7. This class is not a role in feature *TransactionLog*. We create this class, as there is a `LogEntry` array in the `Transaction` class. Furthermore, we create the constructor prototype, because method `transfer` in class `Transaction` creates a new `LogEntry` object.

```
public class LogEntry{

    /*method prototype*/    /
    *@ requires_abs   LogEntryR;
    @  ensures_abs    LogEntryE;
    @  assignable_abs LogEntryA;
    @*/
public LogEntry(Account source, Account destination, int amount){}

}
```

Listing A.7: Class *LogEntry* For Feature Stub of Feature *TransactionLog*

The last class of the feature stub is `LogEntry`. We show it in Listing A.8. This class was not a role in feature *TransactionLog*. We create this class, because method `transfer` of class `Transaction` has two parameters of the type `Account` and we need to match this type access. However, as there is no access to a method or a field from class `Account`, we only need an empty class.

```
public class Account{}
```

Listing A.8: Class *LogEntry* For Feature Stub of Feature *TransactionLog*

B. Appendix

In this appendix, we discuss changes we made on the BankAccount SPL in order to be able to verify its metaproduct by using the contracts instead of Method Inlining. For that purpose, we show the listings of the roles of BankAccount SPL that we needed to change. In the listings of the original version, we already added the **assignable** clauses by hand because they are part of our concept anyway and are not just added because of our goal to use contracts instead of Method Inlining. As showing the complete listings of the changed roles of BankAccount SPL would create many redundancies, we only show the parts of the listings with the changes. We also do not discuss any change made on its own but instead group them in categories.

In Section A.1, we show the listings of the original BankAccount SPL. Subsequently, we discuss the adjustments, we need to make in Section A.2. We conclude the appendix with showing the new roles in Section A.3.

Original Version of BankAccount SPL

```
public class Account {
    public final int OVERDRAFT_LIMIT = 0;

    /*@ public invariant this.balance >= OVERDRAFT_LIMIT; */
    public int balance = 0;

  /*@ ensures balance == 0;
    @ assignable \nothing; @*/
    Account() {
    }

  /*@ requires x != 0;
    @ ensures (!\result ==> balance == \old(balance))
    @   && (\result ==> balance == \old(balance) + x);
    @  assignable balance; @*/
    boolean update(int x) {
        int newBalance = balance + x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = balance + x;
        return true;
    }

  /*@  ensures (!\result ==> balance == \old(balance))
    @   && (\result ==> balance == \old(balance) − x);
    @ assignable balance; @*/
    boolean undoUpdate(int x) {
        int newBalance = balance − x;
        if (newBalance < OVERDRAFT_LIMIT)
            return false;
        balance = newBalance;
        return true;
    }

}
```

Listing B.1: Original Role *Account* of Feature *BankAccount*

```
public class Application {
    /*@ public invariant account != null; */
    public Account account = new Account();

  /*@ requires true;
    @ assignable \nothing; @*/
    void nextDay() {
    }

  /*@ assignable \nothing; @*/
    void nextYear() {
    }
}
```

Listing B.2: Original Role *Application* of Feature *BankAccount*

```
class Account {
    final static int INTEREST_RATE = 2;
    int interest = 0;

  /*@ ensures (balance >= 0 ==> \result >= 0)
    @   && (balance <= 0 ==> \result <= 0);
    @ assignable \nothing; @*/
    int calculateInterest() {
        return balance * INTEREST_RATE / 36500;
    }
}
```

Listing B.3: Original Role *Account* of Feature *Interest*

```
public class Account {
public int[] updates = new int[10];
    public int updateCounter = 0;
    public int[] undoUpdates = new int[10];
    public int undoUpdateCounter = 0;

  /*@ ensures \original;
    @ ensures \result ==> this.updates[this.updateCounter] == x;
    @ ensures \result ==> this.updateCounter ==
    @   (\old(this.updateCounter) + 1) % 10;
    @ ensures !\result ==>this.updateCounter == \old(this.updateCounter);
    @*/
    boolean update(int x){
        if (original(x)){
            updateCounter = (updateCounter + 1) % 10;
            updates[updateCounter] = x;
            return true;
        }
        return false;
    }

  /*@ ensures \original;
    @ ensures \result ==> this.undoUpdates[this.undoUpdateCounter] == x;
    @ ensures \result ==> this.undoUpdateCounter ==
    @   ( \old(this.undoUpdateCounter) + 1 ) % 10;
    @ ensures !\result ==> this.undoUpdateCounter ==
    @   \old(this.undoUpdateCounter); @*/
    boolean undoUpdate(int x){
        if (original(x)){
            undoUpdateCounter = (undoUpdateCounter + 1) % 10;
            undoUpdates[undoUpdateCounter] = x;
            return true;
        }
        return false;
    }
}
```

Listing B.4: Original Role *Account* of Feature *Logging*

```
public class LogEntry {
    /*@ private invariant source != null; @*/
    private Account source;
    /*@ private invariant destination != null;@*/
    private Account destination;
    private int value;

  /*@ requires source != null;
    @ requires destination != null;
    @ requires source != destination;
    @ assignable this.source, this.destination, this.value; @*/
    public LogEntry(Account source,Account destination,int amount){
        this.source = source;
        this.destination = destination;
        this.value = amount;
    }

  /*@ ensures \result != null;
    @ assignable \nothing; @*/
    public Account getSource(){
        return source;
    }

  /*@ ensures \result != null;
    @ assignable \nothing; @*/
    public Account getDestination(){
        return destination;
    }

  /*@ assignable \nothing; @*/
    public int getAmount(){
        return value;
    }
}
```

Listing B.5: Original Role *LogEntry* of Feature *Logging*

```
public class Transaction {
    /*@ requires destination != null && source != null;
      requires source != destination;
      ensures \result ==> (\old(destination.balance) + amount ==
        destination.balance);
      ensures \result ==> (\old(source.balance) - amount ==
        source.balance);
      ensures !\result ==> (\old(destination.balance) ==
        destination.balance);
      ensures !\result ==> (\old(source.balance) ==
        source.balance);
      assignable source.lock, destination.lock, source.balance,
       destination.balance, source.withdraw, destination.withdraw @*/
    public boolean transfer(Account source, Account destination,
    int amount) {
    if (!lock(source, destination)) return false;
        try {
            if (amount <= 0) {
                return false;
            }
            if (!source.update(amount * -1)) {
                return false;
            }
            if (!destination.update(amount)) {
                source.undoUpdate(amount * -1);
                return false;
            }
            return true;
        } finally {
            source.unLock();
            destination.unLock();
        }
    }

    /*@ requires destination != null && source != null;
      requires source != destination;
      ensures \result ==> source.isLocked() && destination.isLocked();
      assignable source.lock, destination.lock;  @*/
    private static synchronized boolean lock(Account source,
    Account destination) {
        if (source.isLocked()) return false;
        if (destination.isLocked()) return false;
        source.lock();
        destination.lock();
        return true;
    }
}
```

Listing B.6: Original Role *Transaction* of Feature *Transaction*

```
public class Transaction {
    /*@ public invariant transactions.length == 10; */
    public LogEntry[] transactions = new LogEntry[10];
    int transactionCounter = 0;

  /*@ requires \original;
      requires \invariant_for(source) && \invariant_for(destination)
        && (\disjoint(source.*, destination.*);
      ensures \original;
      ensures \result ==> ( transactionCounter ==
        (\old(transactionCounter) + 1 ) % 10);
      ensures !\result ==> (transactionCounter ==
        \old(transactionCounter));
      assignable transactionCounter, transactions[*], source.
        updateCounter, source.updates[*], destination.updateCounter,
        destination.updates[*], source.undoUpdateCounter, source.
        undoUpdates[*],  destination.undoUpdateCounter, destination.
        undoUpdates[*];@*/
    boolean transfer(Account source, Account destination, int amount) {
        if (original(source, destination, amount)) {
            transactionCounter = (transactionCounter + 1) % 10;
            transactions[transactionCounter] = new LogEntry(source,
                destination, amount);
            return true;
        }
        return false;
    }
}
```

Listing B.7: Original Role *Transaction* of Feature *TransactionLog*

Adjustments

Normal_Behavior

For all methods, including constructors, we add the JML keyword **`normal_behavior`** to the contract. The keyword is used to define that a method must not throw an exception if its preconditions are fulfilled [Leavens et al., 2006]. We show the keyword in all of the listings below.

Class Invariants

In role `Account` of feature *BankAccount* (see Listing B.1), we remove the invariant **`this`**`.balance >= OVERDRAFT_LIMIT` and add this condition as an **`ensures`** clause to methods `update` and `undoUpdate` in the same role. We show the result in Listing B.8.

We extend the invariant in role `Application` of feature *BankAccount* (see Listing B.2) with `\`**`invariant_for`**`(account)`. The keyword **`invariant_for`** is used to define, that the parameter has to fulfill the invariants of the class it is an instance of [Leavens et al., 2008]. In order for the theorem prover to know which invariants it has to prove we add a second invariant to the role. The invariant **`\typeof`**`(account) ==` **`\type`**`(Account)` states, that the field `account` is of type `Account` [Leavens et al., 2008]. We show the resulting role in Listing B.9.

In role `Account` of feature *Logging*, which we show in Listing B.4, there are two **`int`** array fields that store account updates and two **`int`** fields that save the current position in the array. The arrays have a size of ten elements. To tell the theorem prover their size, we add the invariants `updates.length == 10` and `undoUpdates.length == 10`. Additionally, we define the two counters' values between `0` and `9` with the invariants `updateCounter >= 0 && updateCounter < 10` and `undoUpdateCounter >=0 && undoUpdateCounter < 10`. We also add the invariant `undoUpdates != updates` to the role and show the result in Listing B.11.

In role `Transaction` of feature `TransactionLog`, there is a similar issue as in role `Account`. We show role `Transaction` in Listing B.7. The role includes one array field of the type `LogEntry` that stores the ten last transactions and an **`int`** field that serves as a position pointer for the array. Therefore we also add the invariant `transactionCounter >= 0 && transactionCounter < 10`, so that the counter's value cannot be bigger than the arrays size or smaller than zero. Furthermore, we also add the invariant **`\typeof`**`(transactions) ==` **`\type`**`(LogEntry []` to define that the array has the type `LogEntry`. We present the new role in Listing B.14.

Individual Changes in Method Contracts

For method `calculateInterest`, whose original role we show in Listing B.3, we add the **`ensures`** clause `(`**`\result`**`== calculateInterest())`. The clause is needed for method `estimateInterest` of role `Account` in feature *InterestEstimation*. This method both calls `calculateInterest` in its implementation and its contract. The additional **ensures** clause makes sure, that both calls lead to the same result. We show the new contract for method `calculateInterest` in Listing B.10.

In role `Transaction` of feature `Transaction`(see Listing B.6), we add two **requires** clauses to method `transfer`. First, we add `\invariant_for(source) && \invariant_for(destination)` to define, that the the two method parameters must hold the invariant of their class `Account`. We also add the **requires** clause `\disjoint(source.*, destination.*)`. The keyword `disjoint` means, that its parameters do not share anything on the heap[Albert et al., 2011]. The existing clause **requires** is subsumed by our new clause, therefore we remove it. We show the result in Listing B.13. We add the perform the same adjustments for the contract of method `lock` of the same role.

Finally, we change the contract of constrcutor of role `LogEntry`. We show the the original role in Listing B.5. We add an **ensures** clause defining that the constructor sets the three fields of the class to ensure callers that the fields are set. We show the new version of the role in Listing B.12.

New Version of BankAccount SPL

```
public class Account {
    [...]
    /*@ public normal_behavior
    @ requires x != 0;
    @ ensures ((\old(balance) + x < OVERDRAFT_LIMIT)  <==> !\result);
    @ ensures (!\result ==> balance == \old(balance))
    @ ensures (\result ==> balance == \old(balance) + x);
    @ assignable balance; @*/
    boolean update(int x) {
    [...]
    }

    /*@  public normal_behavior
    @ ensures ((\old(balance) - x < OVERDRAFT_LIMIT) <==> !\result);
    @ ensures (!\result ==> balance == \old(balance))
        && (\result ==> balance == \old(balance) - x);
    assignable balance; @*/
    boolean undoUpdate(int x) {
        [...]
    }
}
```

Listing B.8: New Role *Account* of Feature *BankAccount*

```
public class Application {
  /*@ public invariant (account != null) && \invariant_for(account); @*/
  /*@ public invariant \typeof(account) == \type(Account);*/
    public Account account = new Account();
    [...]
}
```

Listing B.9: New Role *Application* of Feature *BankAccount*

```
class Account {
    final static int INTEREST_RATE = 2;
    int interest = 0;

  /*@ public normal_behavior
    @ ensures (balance >= 0 ==> \result >= 0)
    @   && (balance <= 0 ==> \result <= 0);
    @ ensures \result == calculateInterest();
    @ assignable \nothing; @*/
    int calculateInterest() {
        return balance * INTEREST_RATE / 36500;
    }
}
```

Listing B.10: New Role *Account* of Feature *Interest*

```
public class Account {

/*@ public invariant updates.length == 10; @*/
/*@ public invariant updateCounter >= 0 && updateCounter < 10; @*/
/*@ public invariant undoUpdates.length == 10; @*/
/*@ public invariant undoUpdates != updates; @*/
/*@ public invariant undoUpdateCounter >= 0 && undoUpdateCounter < 10;@*/
    public int[] updates = new int[10];
    public int updateCounter = 0;
    public int[] undoUpdates = new int[10];
    public int undoUpdateCounter = 0;

  /*@ public normal_behavior
    @ ensures \original;
    @ ensures \result ==> this.updates[this.updateCounter] == x;
    @ ensures \result ==> this.updateCounter ==
    @       ( \old(this.updateCounter) + 1 ) % 10;
    @ ensures !\result ==> this.updateCounter ==
    @       \old(this.updateCounter);
    @ assignable updateCounter, updates[*]; @*/
    boolean update(int x){
        [...]
    }

  /*@ public normal_behavior
    @ ensures \original;
    @ ensures \result ==> this.undoUpdates[this.undoUpdateCounter] == x;
    @ ensures \result ==> this.undoUpdateCounter ==
    @       ( \old(this.undoUpdateCounter) + 1 ) % 10;
    @ ensures !\result ==> this.undoUpdateCounter ==
    @       \old(this.undoUpdateCounter);
    @ assignable undoUpdateCounter, undoUpdates[*]; @*/
    boolean undoUpdate(int x){
        [...]
    }
}
```

Listing B.11: New Role *Account* of Feature *Logging*

```
public class LogEntry {
    /*@ private invariant source != null;@*/
    private Account source;
    /*@ private invariant destination != null;@*/
    private Account destination;
    private int value;

  /*@ public normal_behavior
    @ requires source != null;
    @ requires destination != null;
    @ requires source != destination;
    @ ensures this.source == source && this.destination
    @   == destination && this.value == amount;
    @ assignable this.source, this.destination, this.value; @*/
    public LogEntry(Account source,Account destination,int amount){
        this.source = source;
        this.destination = destination;
        this.value = amount;
    }
    [...]
}
```

Listing B.12: New Role *LogEntry* of Feature *Logging*

```
public class Transaction {

  /*@ public normal_behavior
      requires destination != null && source != null;
      requires \invariant_for(source) && \invariant_for(destination)
        && (\disjoint(source.*, destination.*);
      ensures \result ==> (\old(destination.balance) + amount ==
        destination.balance);
      ensures \result ==> (\old(source.balance) - amount ==
        source.balance);
      ensures !\result ==> (\old(destination.balance) ==
        destination.balance);
      ensures !\result ==> (\old(source.balance) == source.balance);
      assignable source.lock, destination.lock, source.balance,
        destination.balance, source.withdraw, destination.withdraw @*/
    public boolean transfer(Account source, Account destination,
    int amount) {
        [...]
    }

  /*@ public normal_behavior
      requires destination != null && source != null;
      requires \invariant_for(source) && \invariant_for(destination)
        && (\disjoint(source.*, destination.*);
      ensures \result ==> source.isLocked() && destination.isLocked();
      assignable source.lock, destination.lock;  @*/
    private static synchronized boolean lock(Account source,
    Account destination) {
        [...]
    }
}
```

Listing B.13: New Role *Transaction* of Feature *Transaction*

```
public class Transaction {
/*@ public invariant transactions.length == 10; */
/*@ public invariant \typeof(transactions) == \type(LogEntry []);@*/
    public LogEntry[] transactions = new LogEntry[10];

/*@ public invariant transactionCounter >= 0
    && transactionCounter < 10;@*/
    int transactionCounter = 0;

  /*@ public normal_behavior
    @ requires \original;
    @ ensures \original;
    @ ensures \result ==> ( transactionCounter ==
    @       (\old(transactionCounter) + 1 ) % 10);
    @ ensures !\result ==> (transactionCounter ==
    @       \old(transactionCounter));
    @ assignable transactionCounter, transactions[*], source.
    @   updateCounter, source.updates[*], destination.updateCounter,
    @   source.undoUpdateCounter, source.undoUpdates[*],
    @   destination.updates[*], destination.undoUpdateCounter,
    @   destination.undoUpdates[*];@*/
    boolean transfer(Account source, Account destination, int amount) {
        [...]
    }
}
```

Listing B.14: New Role *Transaction* of Feature *TransactionLog*

Bibliography

Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P. H., and Ulbrich, M. (2014). The KeY platform for verification and analysis of Java programs. In Giannakopoulou, D. and Kroening, D., editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, Lecture Notes in Computer Science. Springer. To appear. (cited on Page 43)

Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., and Román-Díez, G. (2011). Verified resource guarantees using costa and key. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 73–76. ACM. (cited on Page 87)

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg. (cited on Page 1, 5, 6, and 8)

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *J. Object Technology (JOT)*, 8(5):49–84. (cited on Page 8)

Apel, S., Kästner, C., Größlinger, A., and Lengauer, C. (2010a). Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300. (cited on Page 17 and 67)

Apel, S., Kästner, C., and Lengauer, C. (2008). Feature featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 101–112, New York, NY, USA. ACM. (cited on Page 17 and 67)

Apel, S., Kästner, C., and Lengauer, C. (2013b). Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79. (cited on Page 2, 8, 42, and 43)

Apel, S., Kolesnikov, S., Liebig, J., Kästner, C., Kuhlemann, M., and Leich, T. (2012). Access control in feature-oriented programming. *Science of Computer Programming (SCP)*, 77(3):174–187. (cited on Page 43)

Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., and Garvin, B. (2013c). Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, pages 1–8, New York, NY, USA. ACM. (cited on Page 13)

Apel, S. and Lengauer, C. (2008). Superimposition: A language-independent approach to software composition. In *Proc. Int'l Symposium Software Composition -SC*, pages 20–35. (cited on Page 8)

Apel, S., Rhein, A. v., Thüm, T., and Kästner, C. (2013d). Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409. (cited on Page 16)

Apel, S., Rhein, A. v., Wendler, P., Größlinger, A., and Beyer, D. (2013e). Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491, Piscataway. IEEE. (cited on Page 2, 30, 31, 56, and 68)

Apel, S., Scholz, W., Lengauer, C., and Kästner, C. (2010b). Detecting dependences and interactions in feature-oriented design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170, Washington, DC, USA. IEEE. (cited on Page 16)

Apel, S., Speidel, H., Wendler, P., Rhein, A. v., and Beyer, D. (2011). Feature-aware verification. (cited on Page 67)

Barnett, M., Fähndrich, M., Leino, K. Rustan M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and verification: The spec# experience. *Comm. ACM*, 54(6):81–91. (cited on Page 11)

Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, volume 3714, pages 7–20, Berlin, Heidelberg. Springer. (cited on Page 6 and 7)

Beckert, B., Hähnle, R., and Schmitt, P. (2007). *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, Berlin, Heidelberg, New York, London. (cited on Page 1, 2, 3, 12, 13, 38, and 43)

Benduhn, F. (2012). Contract-aware feature composition. Bachelor's thesis, University of Magdeburg, Germany. (cited on Page 42 and 43)

Bettini, L., Damiani, F., and Schaefer, I. (2014). Implementing type-safe software product lines using parametric traits. *Science of Computer Programming (SCP).* (cited on Page 67)

Beuche, D. (2003). *Composition and Construction of Embedded Software Families.* PhD thesis, University of Magdeburg, Germany. (cited on Page 5)

Bontemps, Y., Heymans, P., Schobbens, P.-Y., and Trigaux, J.-C. (2004). Semantics of feature diagrams. In *Proc. Int'l Workshop Software Variability Management for Product Derivation - Towards Tool Support (SVMPD).* (cited on Page 7)

Bruns, D., Klebanov, V., and Schaefer, I. (2011). Verification of software product lines with delta-oriented slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *LNCS*, pages 61–75, SpringerA. Springer. (cited on Page 68)

Bubel, R., Din, C., and Hähnle, R. (2010). Verification of variable software: An experience report. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, Karlsruhe and Germany. Technical Report 2010-13, Department of Informatics, Karlsruhe Institute of Technology. (cited on Page 67)

Bubel, R., Hähnle, R., and Pelevina, M. (2014). Fully abstract operation contracts. In *Proceedings 6th International Symposium On Leveraging Applications of Formal Methods*, LNCS. Springer. to appear. (cited on Page 14, 22, 51, 52, and 69)

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J., Leavens, G. T., Leino, K. Rustan M., and Poll, E. (2005). An overview of JML tools and applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232. (cited on Page 11 and 12)

Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141. (cited on Page 16)

Calder, M. and Miller, A. (2006). Feature interaction detection by pairwise analysis of ltl properties—a case study. *Formal Methods in System Design*, 28(3):213–261. (cited on Page 16)

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010). Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, NY, USA. ACM. (cited on Page 67)

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, USA. (cited on Page 5)

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

Damiani, F., Dovland, J., Johnsen, E. B., Owe, O., Schäfer, I., and Chieh Yu, I. (2012). A transformational proof system for delta-oriented programming: Proceedings of the 16th international software product line conference. In Santana de Almeida, Eduardo, editor, *Proceedings of the 16th International Software Product Line Conference*, volume 2, pages 53–60, New York and NY and USA. ACM. (cited on Page 68)

Delaware, B., Cook, W., and Batory, D. (2011). Product lines of theorems. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 595–608, New York, NY, USA. ACM. (cited on Page 68)

Delaware, B., d. S. Oliveira, Bruno C., and Schrijvers, T. (2013). Meta-theory à la carte. In *Proc. Conf. Principles of Programming Languages (POPL)*, pages 207–218, New York, NY, USA. ACM. (cited on Page 68)

Ehrenberger, W. (2002). *Software-Verifikation: Verfahren für den Zuverlässigkeitsnachweis von Software.* Hanser, München, Wien. (cited on Page 12)

Fantechi, A. and Gnesi, S. (2008). Formal modeling for product families engineering. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 193–202, Washington. IEEE Computer. (cited on Page 67)

Gondal, A., Poppleton, M., and Butler, M. (2011). Composing event-b specifications: Case-study experience. In *Proc. Int'l Symposium Software Composition (SC)*, pages 100–115, Berlin, Heidelberg. Springer. (cited on Page 68)

Haber, A., Rendel, H., Rumpe, B., and Schaefer, I. (2012). Evolving delta-oriented software product line architectures. In *Monterey*, volume 7539 of *LNCS*, pages 183–208, Berlin, Heidelberg, New York, London. Springer. (cited on Page 8)

Hähnle, R. and Schaefer, I. (2012). A Liskov principle for delta-oriented programming. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 1, pages 32–46, Berlin, Heidelberg. Springer Berlin Heidelberg. (cited on Page 14 and 69)

Hähnle, R., Schaefer, I., and Bubel, R. (2013a). Reuse in software verification by abstract method calls. In *Proc. Int'l Conf. Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 300–314, Berlin, Heidelberg. Springer. (cited on Page 14)

Hähnle, R., Schaefer, I., and Bubel, R. (2013b). Reuse in software verification by abstract method calls: 24th international conference on automated deduction, lake placid, ny, usa, june 9-14, 2013. proceedings. In Bonacina, M. P., editor, *Automated Deduction – CADE-24*, pages 300–314, Berlin and Heidelberg. Springer Berlin Heidelberg. (cited on Page 69)

Harhurin, A. and Hartmann, J. (2008). Towards consistent specifications of product families. In *Proc. Int'l Symposium Formal Methods (FM)*, pages 390–405, Berlin, Heidelberg. Springer. (cited on Page 67)

Hatcliff, J., Leavens, G. T., Leino, K. Rustan M., Müller, P., and Parkinson, M. (2012). Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58. (cited on Page 1 and 11)

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580. (cited on Page 11)

Istoan, P. (2013). *Methodology for the Derivation of Product Behaviour in a Software Product Line*. PhD thesis, Université Rennes 1, Luxembourg. (cited on Page 17)

Jézéquel, J.-M. and Meyer, B. (1997). Design by contract: The lessons of ariane. *IEEE Computer*, 30(1):129–130. (cited on Page 1 and 11)

Jörges, S., Lamprecht, A.-L., Margaria, T., Schaefer, I., and Steffen, B. (2012). A constraint-based variability modeling framework. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14(5):511–530. (cited on Page 67)

Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65. (cited on Page 5 and 6)

Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012). Type checking annotation-based product lines. *Trans. Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:39. (cited on Page 17 and 67)

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, NY, USA. ACM. (cited on Page 67)

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, Berlin, Heidelberg, New York, London. Springer. (cited on Page 8)

Kolesnikov, S., Rhein, A. v., Hunsen, C., and Apel, S. (2013). A comparison of product-based, feature-based, and family-based type checking. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 115–124, New York, NY, USA. ACM. (cited on Page 2, 16, and 17)

Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38. (cited on Page 12 and 86)

Leavens, G. T. and Cheon, Y. (2006). Design by contract with JML. (cited on Page 12)

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., and Chalin, P. (2008). Jml reference manual. (cited on Page 36 and 86)

Meinicke, J. (2013). JML-based verification for feature-oriented programming. Bachelor's thesis, University of Magdeburg, Germany. (cited on Page 15, 27, 28, 30, 31, 34, 35, 36, 38, 39, 51, 56, and 68)

Meinicke, J., Thüm, T., Schöter, R., Benduhn, F., and Saake, G. (2014). An overview on analysis tools for software product lines. In *Workshop on Software Product Line Analysis Tools (SPLat)*, pages 94–101, New York, NY, USA. ACM. (cited on Page 2)

Meyer, B. (1992). Applying design by contract. *IEEE Computer*, 25(10):40–51. (cited on Page 11)

Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 127–136, New York, NY, USA. ACM. (cited on Page 8)

Necula, G. C. (1997). Proof-carrying code. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 106–119, New York, NY, USA. ACM. (cited on Page 38)

Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., and Guo, J. (2013a). Feature-oriented software evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 17:1–17:8, New York, NY, USA. ACM. (cited on Page 26 and 69)

Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wasowski, A., and Borba, P. (2013b). Coevolution of variability models and related artifacts: A case study from the Linux kernel. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 91–100, NY, USA. ACM. (cited on Page 26)

Pelevina, M. (2014). Realization and extension of abstract operation contracts for program logic. Bachelor's thesis, TU Darmstadt, Germany. (cited on Page 22)

Pohl, K., Böckle, G., and van der Linden, Frank J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg, New York, London. (cited on Page 5)

Praast, M. (2014). Effiziente kodierung von variabilität in spezifikationen. Master's thesis, University of Magdeburg, Germany. In German. (cited on Page 51)

Prehofer, C. (1997). 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. Springer. (cited on Page 8)

Proksch, F. and Krüger, S. (2014). Tool support for contracts in FeatureIDE. Technical Report FIN-001-2014, School of Computer Science, University of Magdeburg, Germany. (cited on Page 42)

Sabouri, H. and Khosravi, R. (2010). An effective approach for verifying product lines in presence of variability models. In Botterweck, G., Jarzabek, S., Kishi, T., Lee, J., and Livengood, S., editors, *Proc. Int'l Workshop Formal Methods and Analysis* *in Software Product Line Engineering (FMSPLE)*, pages 113–120, UK. Lancaster University. (cited on Page 67)

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, volume 6287 of *LNCS*, pages 77–91, Berlin, Heidelberg. Springer. (cited on Page 8)

Scholz, W., Thüm, T., Apel, S., and Lengauer, C. (2011). Automatic detection of feature interactions using the Java Modeling Language: An experience report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8, New York, NY, USA. ACM. (cited on Page 16)

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014a). A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45. (cited on Page 1, 2, 13, 17, 27, 56, and 67)

Thüm, T., Apel, S., Zelend, A., Schröter, R., and Möller, B. (2013). Subclack: Feature-oriented programming with behavioral feature interfaces. In *Proc. Workshop MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 1–8, New York, NY, USA. ACM. (cited on Page 69)

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014b). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming (SCP)*, 79(0):70–85. (cited on Page 2 and 42)

Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., Rhein, A. v., and Saake, G. (2014c). Potential synergies of theorem proving and model checking for software product lines. *Proc. Int'l Software Product Line Conf. (SPLC). ACM.* (cited on Page 2, 51, and 68)

Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012a). Family-based deductive verification of software product lines. In *International Conference on Generative Programming and Component Engineering*, pages 11–20, New York, NY, USA. ACM. (cited on Page 13, 15, 27, 28, 30, 34, 39, 51, 56, 60, 66, and 68)

Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. (2011). 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. IEEE Computer. (cited on Page 68)

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. (2012b). Applying design by contract to feature-oriented programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212, pages 255–269, Berlin, Heidelberg. Springer. (cited on Page 12 and 15)

Wampfler, D. (2007). Aspect-oriented design principles: Lessons from object-oriented design. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages I6:1–I6:10, New York, NY, USA. ACM. (cited on Page 8)

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

Magdeburg, den 18.November 2014