Family-Based Deductive Verification of Software Product Lines

Thomas Thüm
University of Magdeburg
Germany

Ina Schaefer
University of Braunschweig
Germany

Sven Apel
University of Passau
Germany

Martin Hentschel
University of Darmstadt
Germany

ABSTRACT

A software product line is a set of similar software products that share a common code base. While software product lines can be implemented efficiently using feature-oriented programming, verifying each product individually does not scale, especially if human effort is required (e.g., as in interactive theorem proving). We present a family-based approach of deductive verification to prove the correctness of a software product line efficiently. We illustrate and evaluate our approach for software product lines written in a feature-oriented dialect of Java and specified using the Java Modeling Language. We show that the theorem prover KeY can be used off-the-shelf for this task, without any modifications. Compared to the individual verification of each product, our approach reduces the verification time needed for our case study by more than 85 %.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Reliability, Verification

Keywords

Product-line analysis, software product lines, program families, deductive verification, theorem proving

1. INTRODUCTION

A challenging task in software engineering is to develop reuse strategies for software. Software product-line engineering aims at reuse across similar software products [13, 14]. A *software product line* is a set of software-intensive systems that share a common code base [13]. The products of a software product line are distinguished in terms of features. A *feature* is an end-user-visible program characteristic [20]. Generative programming is an approach to implement product lines such that each product can be derived automatically based on a feature selection [14]. We illustrate our approach using feature-oriented programming, an instance of generative programming, in which features are made explicit in design and code [29, 8].

Software reuse increases the dependability of software, if the behavior of reused software is specified explicitly [19]. *Design by contract* is a technique to formally specify the behavior of software along with source code [27]. In essence, programmers annotate methods with contracts. A *contract* consists of *preconditions* the caller needs to ensure and *postconditions* the caller can rely on. The *Java Modeling Language* (JML) supports contracts in Java, which can be used for documentation generation, runtime assertion checking, and deductive verification [11]. We focus on deductive verification of JML contracts using the theorem prover KeY [9].

Software product lines challenge existing verification techniques [33, 24, 6, 12, 32]. A simple strategy to verify a software product line is to generate and verify each product separately—pursuing a *product-based strategy* [33]. But this strategy often leads to highly redundant verification tasks, and scales only for product lines with a small number of products. Especially for interactive theorem proving, a product-based strategy does not involve only redundant computations, but also redundant human interaction for completing the verification. Apart from that, it is often not even possible to generate all software products, due to their large number, as for example, for the Linux kernel [31].

A recent idea is to apply verification to the product-line's code base (instead of to all generated products) in order to omit redundant analyses—pursuing a *family-based strategy* [33]. Family-based strategies have been proposed for the analysis and verification of software product lines using type checking and model checking [33]. In principle, there are two family-based approaches: First, a new tool is built or an existing verification tool is adapted to incorporate both the code base and variability [15, 16, 24, 12]. Second, the whole software product line is encoded as a single *metaproduct* in the input language of an existing verification tool using *variability encoding* [28, 6]; the metaproduct simulates the behavior of all individual products of the product line in question. We use the second approach as it does not involve to build new verification tools —which need to be trusted—

and apply variability encoding to program verification by theorem proving. In summary, we make the following contributions:

- We propose the first family-based, deductive verification approach that operates on the product line's code base, and that takes variability into account.
- We discuss and illustrate how to apply variability encoding to programs or libraries; previous work has focused on stand-alone applications only [6].
- We apply variability encoding not only to code, but also to specifications; we translate the specifications given for each feature into a *metaspecification* containing the specification of all products.
- By means of a case study, we demonstrate how to use the theorem prover KeY as-is to verify software product lines and evaluate the efficiency of our approach.

2. BACKGROUND

In this section, we introduce the underlying concepts of our approach. We describe how feature models specify valid combinations of features, and how feature-oriented programming can be used to develop software product lines.

2.1 Feature Model

We distinguish the products of a software product line by means of features, but not all combinations of features may lead to valid products. For example, a software product line may support different platforms such as Linux and Windows. When modeling these platforms as features, it is often invalid to select both features for the same product.

A *feature model* is a hierarchical structure defining all features of a product line and their valid combinations [20]. Each feature may have subfeatures that are either *optional*, *mandatory*, or belong to a group. Common group types are *alternative* (exactly one of the subfeatures needs to be selected) and *or* (at least one of the subfeatures needs to be selected) [7]. Whenever a feature is selected, its parent feature is selected, as well.

Example. As a running example, we use a product line of banking software. Each product of this product line is a software to manage a bank account, but products differ in the individual features they provide. In Figure 1, we illustrate the features and their valid combinations. Feature *BankAccount* provides a rudimentary bank account storing the current balance. Feature *Overdraft* indicates whether the bank allows their customers to withdraw more money from the account than actually available, if the resulting negative balance is within an overdraft limit. Feature *Interest* states whether the customer gets interests, and feature *InterestEstimation* provides a calculation of the expected interest for the current year. Feature *CreditWorthiness* allows the bank to assess whether a customer may get a credit of a certain amount. Finally, feature *DailyLimit* allows the bank to limit the daily withdrawal. All features can be combined arbitrarily except that feature *InterestEstimation* requires feature *Interest*. Hence, the feature model describes 24 valid combinations of features (i.e., 24 products). □

For automated reasoning, feature models can also be represented using *propositional formulas* [7]. A boolean variable

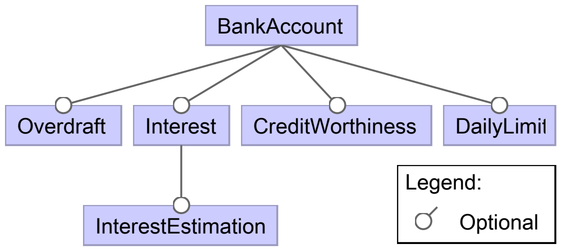

Figure 1: A feature model specifying the valid combinations of features in a product line of banking software.

is used for each feature, and the propositional formula evaluates to true, if and only if the combination of features is valid. Every relation between a feature and its subfeatures is translated into a propositional formula, which are then conjoined to a large formula representing the whole feature model [7]. We give a propositional formula for our example:

$$\begin{aligned}
&\mathit{BankAccount}\\
&\wedge (\mathit{Overdraft} \Rightarrow \mathit{BankAccount})\\
&\wedge \mathit{Interest} \Rightarrow \mathit{BankAccount}\\
&\wedge (\mathit{CreditWorthiness} \Rightarrow \mathit{BankAccount})\\
&\wedge (\mathit{DailyLimit} \Rightarrow \mathit{BankAccount})\\
&\wedge (\mathit{InterestEstimation} \Rightarrow \mathit{Interest})
\end{aligned}$$

2.2 Feature-Oriented Programming

The aim of feature-oriented programming is to enable developers to automatically generate products of a software product line based on a selection of features [29, 8]. A key concept is to make features explicit in design and code in terms of feature modules (one module per feature). In a Java world, a *feature module* is a set of classes and class refinements covering the implementation of a certain feature. A *class refinement* refines an existing class by adding new members such as methods and fields, and by applying method refinements. A *method refinement* overrides an existing method. A software product is generated by composing the feature modules according to a given feature selection [8]. Typically, there is a total order of all feature modules for composition.

Example. In Figure 2, we show the code of three feature modules for our running example (ignore comments in source code for now). Feature module *BankAccount* introduces two standard Java classes. Class `Account` stores the current balance of a bank account and enables to withdraw or deposit money using method `update()`, as long as the balance remains positive. For brevity, class `Application` only stores one object of type `Account` and provides method `nextDay()`, which is assumed to be called once at midnight.

Feature module *Overdraft* refines the value of the constant `OVERDRAFT_LIMIT` in class `Account` to permit the account holder to overdraw his account up to a balance of −5.000 €. Composing this feature module with feature module *BankAccount* replaces the definition of the constant.

Feature module *DailyLimit* refines class `Account` to store the total withdraw of the day. It provides a new implementation of method `update()` calling the original method using keyword `original` (Line 47). When composing all three feature modules, the method introduction from feature *BankAccount* is renamed, and the keyword `original` is replaced by a call to the renamed method. Class `Application` is refined to reset the withdraw every day. □

3. FEATURE-BASED SPECIFICATION

The theorem prover KeY enables a user to verify that a given Java program fulfills its JML specification.[1] As we want to verify software product lines using KeY, we need an approach to specify the intended behavior of the *entire* software product line, not only of a single product. In previous work, we discussed several approaches for the specification of software product lines [35]. Here, we use *explicit contract refinement*, because it is expressive and intuitive for programmers familiar with feature-oriented programming [35]. We illustrate how JML is used for the specification of object-oriented Java programs. Then, we discuss how we extend JML for the specification of feature-oriented Java programs.

3.1 JML for Object-Oriented Programming

The idea of design by contract is to enrich programs with specifications, given in a formal language, that can be processed by tools [27]. In object-oriented programming, design by contract can be used to specify the intended behavior of methods in terms of method contracts and class invariants. A *method contract* is basically a pair of precondition and postcondition. The *precondition* states what the caller needs to ensure and the method can rely on; the *postcondition* states what the method needs to ensure and what the caller can rely on. A *class invariant* formulates properties that the constructor needs to establish and that all methods of a class need to maintain, except those methods explicitly marked as *helper methods* by the programmer.

JML enables the specification of method contracts and class invariants in Java. JML specifications are defined inside special comments to ensure that standard Java tools such as editors and compilers can still be used [25]. The syntax of JML is similar to the syntax of expressions in Java, but contains further keywords identifying preconditions, postconditions, and invariants, amongst others.

Example. We give examples of invariants and postconditions in the feature module *BankAccount* shown in Figure 2. This feature module contains standard Java classes with JML annotations. In Lines 3–4, we define an invariant stating that the balance is within the overdraft limit. In Lines 7–10, we define a postcondition for method `update` specifying how the balance is altered. Line 21 defines an invariant stating that field `account` must always be initialized. □

3.2 JML for Feature-Oriented Programming

In feature-oriented programming, a feature module may also contain class and method refinements, raising the question how to write a JML specification for them. According to explicit contract refinement [35], a refinement of a class may introduce new invariants to this class and a method refinement may refine existing contracts in a similar manner as method implementations can be refined. We extend

[1]More precisely, KeY enables the verification of Java Card programs (a subset of Java programs) with some extensions such as support for strings.

```
 1 class Account {                              BankAccount
 2   final static int OVERDRAFT_LIMIT = 0;
 3   //@ invariant balance >=
 4   //@   OVERDRAFT_LIMIT;
 5   int balance = 0;
 6   /*@
 7    @ ensures (!\result ==>
 8    @     balance == \old(balance))
 9    @   && (\result ==>
10    @     balance == \old(balance) + x);
11    @*/
12   boolean update(int x) {
13     int newBalance = balance + x;
14     if (newBalance < OVERDRAFT_LIMIT)
15       return false;
16     balance = newBalance;
17     return true;
18   }
19 }
20 class Application {
21   //@ invariant account != null;
22   Account account = new Account();
23   void nextDay() {}
24 }
```

```
25 refines class Account {                        Overdraft
26   final static int OVERDRAFT_LIMIT =
27     -5000;
28 }
```

```
29 refines class Account {                       DailyLimit
30   final static int DAILY_LIMIT = -1000;
31   //@ invariant withdraw >= DAILY_LIMIT;
32   int withdraw = 0;
33   /*@
34    @ ensures \original
35    @   && (!\result ==>
36    @     withdraw == \old(withdraw))
37    @   && (\result ==>
38    @     withdraw<=\old(withdraw));
39    @*/
40   boolean update(int x) {
41     int newWithdraw = withdraw;
42     if (x < 0)  {
43       newWithdraw += x;
44       if (newWithdraw < DAILY_LIMIT)
45         return false;
46     }
47     if (!original(x))
48       return false;
49     withdraw = newWithdraw;
50     return true;
51   }
52 }
53 refines class Application {
54   //@ ensures account.withdraw == 0;
55   void nextDay() {
56     original();
57     account.withdraw = 0;
58   }
59 }
```

Figure 2: Three feature modules of our bank account product line: *BankAccount*, *Overdraft*, and *DailyLimit*.

JML by the keyword `original` that may be used only in preconditions and postconditions of method refinements. When composing feature modules and their JML specification, keyword `original` is replaced by the precondition or postcondition of the original method, respectively.

Example. The features *Overdraft* and *DailyLimit* in Figure 2 refine the specification of feature module *BankAccount*. Feature *Overdraft* refines the invariant shown in Lines 3–4 indirectly, by changing the constant `OVERDRAFT_LIMIT`. Thus, the selection of feature *Overdraft*, does not only influence program execution, but also the specification. Feature *DailyLimit* adds an invariant to class `Account` stating that the total withdrawal of the day is within the limit (Line 31). The postcondition of the refinement of method `update()` states that the total withdrawal of the day is updated correctly, and that the postcondition of the original method is maintained (Lines 34–38). The refinement of method `nextDay()` is annotated with a postcondition ensuring that the withdrawal has been reset (Line 54). □

4. FAMILY-BASED VERIFICATION

The underlying idea of our approach is to generate a metaproduct for a given software product line that contains the implementation and specification of *every* feature, and thus can simulate every product. We propose to verify the metaproduct (family-based) instead of verifying all products separately (product-based). From the verification of the metaproduct, we can conclude that every product of the product line fulfills its specification. Similar approaches were successful for model checking [28, 6].

For verification, we need to trust in the verification tool. Hence, we want to use the theorem prover KeY as-is, so the metaproduct must be a Java program with JML specifications (nonetheless representing all products of the software product line). This way, we can rely on an existing theorem prover known from single-system engineering, but we have to guarantee that the metaproduct is constructed correctly.

In the following, we describe how to generate a metaproduct from a set of feature modules, feature specifications, and a feature model.

4.1 Variability Encoding of Feature Modules

The transformation of feature modules into a metaproduct (a.k.a. product simulator) was proposed by Apel et al. [6], which we adapt for our purpose. For each feature, a new variable is introduced, indicating whether the feature is selected or not. The *feature variable* is then used to switch between different behaviors depending on the selection of the feature. In feature-oriented programming, we select and compose features before compile time, but the metaproduct contains the code of the whole product line and uses dynamic branching (i.e., Java's `if-then`-blocks) to cover the behavior of all feature combinations.

All feature modules of the product line are composed by merging class introductions with their respective class refinements (the algorithm is described elsewhere in detail [6]). The non-trivial case is how to handle method refinements, because the method body of a refined method depends on the feature selection. In a nutshell, we generate a distinct method for every method introduction and method refinement, where refined methods are renamed to distinguish them in the resulting metaproduct. At the beginning of each method refinement, we add a dynamic branch checking whether the corresponding feature of the refinement is selected or not. If the feature is selected, we continue with the implementation that this feature has introduced for the method, and if not, we call the original method (i.e., the previous method refinement, if existent, or the method introduction, otherwise).

Example. Figure 3 illustrates an excerpt of the metaproduct generated based on the feature modules shown in Figure 2. Class `Account` contains the fields `balance`, `withdraw`, `OVERDRAFT_LIMIT`, and `DAILY_LIMIT`, as defined in respective feature modules. Method `update()`, defined in feature module *BankAccount*, is renamed (Lines 14–21). In the refinement of method `update()`, defined originally in feature module *DailyLimit*, a new branching statement is added as first statement (Lines 50–63). If feature *DailyLimit* is not selected, the original method is called and then the method returns. Otherwise, the method is executed as defined in feature module *DailyLimit*. □

Similarly to Kästner et al. [21], we assume type uniformity for all feature modules. That is, (a) all valid combinations of feature modules are well-typed *and* (b) the composition of *all* feature modules is well-typed. The first condition is necessary as only well-typed programs can be verified. The second condition is necessary as mutually exclusive features may introduce incompatible classes or class refinements, which may cause type errors in the metaproduct. For example, two mutually exclusive feature modules may introduce a field to a certain class with the same name but of incompatible types. In this case, the metaproduct is ill-typed, even though every valid product is well-typed. The problem can be solved by renaming based on a variability-aware type system [2]. However, mutually exclusive features introducing incompatible types are rare in feature-oriented product lines [3], and did not occur in our case study.

Furthermore, we assume that there is no *name shadowing* in the resulting metaproduct. Name shadowing can occur when a field of a class has the same name as a local variable (lexical shadowing) or if a field is defined in a class and its superclass (inheritance-based shadowing). For an example consider Figure 2 again: if method `nextDay()` in feature module *DailyLimit* would declare a variable `account`, then the statement in Line 57 would access the local variable instead of field `account`. With name shadowing, a certain statement can have different meanings depending on the feature selection, but only one meaning in the metaproduct. That is, the metaproduct does not cover the behavior of all products. Fortunately, name shadowing can be detected statically. However, it is part of future work to handle name shadowing in metaproducts correctly.

4.2 Variability Encoding of Specifications

Since we want to formally verify whether the metaproduct fulfills its specification, we also need to transform the specifications given for each feature module into one large *metaspecification*. The metaspecification must be a valid JML specification with respect to the metaproduct, such that we can use the theorem prover KeY for verification.

Method contracts are always generated together with their method introduction or method refinement. Furthermore, class invariants are also included in their respective class.

```
 1 class Account {
 2 
 3 // SOURCE CODE FROM FEATURE BankAccount
 4 
 5   //@ invariant balance >=
 6   //@   OVERDRAFT_LIMIT;
 7   int balance = 0;
 8   /*@
 9    @ ensures (!\result ==>
10    @      balance == \old(balance))
11    @    && (\result ==>
12    @      balance == \old(balance) + x);
13    @*/
14   boolean /*@ helper @*/
15   update$$BankAccount(int x) {
16     int newBalance = balance + x;
17     if (newBalance < OVERDRAFT_LIMIT)
18       return false;
19     balance = newBalance;
20     return true;
21   }
22 
23 // SOURCE CODE FROM FEATURE Overdraft
24 
25   final static int OVERDRAFT_LIMIT =
26     FeatureModel.overdraft ? −5000 : 0;
27 
28 // SOURCE CODE FROM FEATURE DailyLimit
29 
30   final static int DAILY_LIMIT = −1000;
31   //@ invariant FeatureModel.dailyLimit
32   //@     ==> withdraw >= DAILY_LIMIT;
33   int withdraw = 0;
34   /*@
35    @ ensures !FeatureModel.dailyLimit==>
36    @    (!\result ==>
37    @      balance == \old(balance))
38    @    && (\result ==>
39    @      balance == \old(balance)+x);
40    @ ensures FeatureModel.dailyLimit ==>
41    @    ((!\result ==>
42    @        balance == \old(balance))
43    @      && (\result ==>
44    @        balance == \old(balance)+x))
45    @    && (!\result ==>
46    @      withdraw == \old(withdraw))
47    @    && (\result ==>
48    @      withdraw <= \old(withdraw));
49    @*/
50   boolean update(int x) {
51     if (!FeatureModel.dailyLimit)
52       return update$$BankAccount(x);
53     int newWithdraw = withdraw;
54     if (x < 0)  {
55       newWithdraw += x;
56       if (newWithdraw < DAILY_LIMIT)
57         return false;
58     }
59     if (!update$$BankAccount(x))
60       return false;
61     withdraw = newWithdraw;
62     return true;
63   }
64 
65 }
```

Figure 3: Metaproduct including metaspecification for the class Account defined in feature *BankAccount* and its two class refinements from the features *Overdraft* and *DailyLimit* as shown in Figure 2.

Thus, every method of the metaproduct will have the same specification as defined in the respective feature.

The resulting specifications are not yet valid JML specifications as they may contain references to refined preconditions or postconditions using the keyword `original` (e.g., as in Figure 2, Line 34). Hence, we replace every occurrence of `original` by the refined precondition or postcondition, respectively. The order of replacement is important; it is necessary to start with method introductions and then to continue with every method refinement in the refinement chain (recall, there is a total order of feature modules).

The specifications retrieved using this procedure are valid JML specifications, but do not consider variability (i.e., specifications do not depend on the feature selection). However, our approach for the specification of feature modules actually supports the definition of specifications that are only satisfied when a certain feature is selected. Hence, we propose to use the feature variables introduced for implementation also in the specifications. The resulting metaproduct does then also contain a metaspecification, against which we can check the correctness of the metaproduct.

Before replacing occurrences of the keyword `original`, we need to apply some additional rewritings. First, every class invariant *inv* introduced in feature module f is rewritten to the implication $f \Rightarrow inv$ to indicate that the invariant *inv* is assumed only to hold if feature f is selected. Similarly, every precondition *pre* and postcondition *post* is replaced by $f \Rightarrow pre$ and $f \Rightarrow post$, respectively. A simple optimization is to keep each specification as-is that is specified in a core feature. A *core feature* is a feature that is included in every product and thus its specification must be fulfilled by every product.

But so far, the semantics of method contracts in the metaproduct is not the same as for the feature modules. For instance, given that we have a method introduction m_i in feature module f and a method refinement m_r for m_i in feature module g. If the feature variable for g has the value `false`, precondition and postcondition of the resulting method always evaluate to `true`. Hence, there are essentially no precondition and postcondition the method and caller can rely on, but we may have defined precondition and postcondition for m_i in feature module f. Thus, it is also necessary to specify the contract if a certain feature is *not* selected.

Consequently, we propose to generate method contracts as follows. Given precondition *pre* and postcondition *post* of a certain method refinement defined in feature module f, we transform the precondition into $(f \Rightarrow pre) \wedge (\neg f \Rightarrow pre')$ and the postcondition into $(f \Rightarrow post) \wedge (\neg f \Rightarrow post')$, in which pre' refers to the precondition and $post'$ to the postcondition of the original method. If there is no original method (i.e., for method introductions), only precondition $f \Rightarrow pre$ and postcondition $f \Rightarrow post$ are generated for the metaproduct.

Example. The metaproduct shown in Figure 3 also contains its metaspecification. The invariant from feature module *DailyLimit* is transformed into an implication stating that it is established only if feature *DailyLimit* is selected (Lines 31–32). In contrast, the invariant from feature module *BankAccount* is not transformed into an implication, because feature *BankAccount* is a core feature and its specification must be fulfilled by all products (Lines 5–6). For the same reason, the contract of the method introduction `update()` is copied as-is. The only change is that this method is annotated with the keyword `helper` to indicate that it does not need to fulfill class invariants. Note that all renamed methods must be marked as helper methods, because they are not intended to fulfill all class invariants.

The transformation for the contract of method refinement `update()`, defined in feature module *DailyLimit*, is more complex. We generate two postconditions stating the behavior for products containing the feature *DailyLimit* and those not containing it. If feature *DailyLimit* is not selected, the method guarantees the original postcondition as defined in feature module *BankAccount*. Hence, we copy the postcondition from method `update$$BankAccount()` (Lines 35–39). Otherwise, if feature *DailyLimit* is selected, the method guarantees the contract as defined in feature module *DailyLimit*, in which we need to replace the keyword `original` by the postcondition from method `update$$BankAccount()` (Lines 40–48). □

4.3 Variability Encoding of Feature Models

We illustrated how to transform feature modules and their specifications into a metaproduct, but we ignored how to actually initialize the variables representing each feature with values `true` or `false`. The idea is to initialize the feature variables nondeterministically at runtime. Consequently, a static verification tool cannot make assumptions on the feature selection and thus verifies all possible combinations [6].

In Java, the point where the program starts may not be unique. For example, a Java program may have several main methods in different classes. Alternatively, we may also want to verify a Java library and basically every static method or constructor may be the entry point of the program. Our solution is to use Java's class loading for nondeterministic initialization of feature variables. We add a class containing all feature variables. When this class is accessed for the first time, the class is loaded and random values are assigned to each feature variable, to make sure that the values are arbitrary. Hence, verification tools cannot rely on particular values and *all* feature combinations are verified.

Finally, our metaproduct should cover only the products of our software product line defined in the corresponding feature model. So far, we would try to verify all combinations of features, which may fail as certain features are not designed or intended to be compatible. To overcome this limitation, we check whether the random feature selection is valid by transforming the feature model into an invariant consisting of a propositional formula.

Example. In Figure 4, we illustrate the class that encodes our example feature model. Feature variables are modeled as static fields and initialized in a static constructor using random values. The dependencies between features are modeled using a propositional formula, which is encoded as a Java expression (Lines 16–21). If the random initialization is invalid according to the feature model, the program is terminated (Lines 11–12). Otherwise, we can assume the feature model as an invariant (Line 2). □

5. EVALUATION

We applied our approach to the verification of a product line of bank accounts, which we discussed already partially in previous sections as our running example.[2] Next, we describe our tool setting, the actual case study, and our results.

[2]The source code of all feature modules is available online: `http://spl2go.cs.ovgu.de/`

```
 1 public class FeatureModel {
 2   //@ static invariant fm();
 3   public final static boolean
 4     bankAccount, overdraft,
 5     interest, interestEstimation,
 6     creditWorthiness, dailyLimit;
 7
 8   static {
 9     bankAccount = random();
10     //initialization of other variables
11     if (!fm())
12       System.exit(1);
13   }
14
15   /*@ pure @*/ boolean fm() {
16     return bankAccount
17     && (!overdraft||bankAccount)
18     && (!interest||bankAccount)
19     && (!creditWorthiness||bankAccount)
20     && (!dailyLimit||bankAccount)
21     && (!interestEstimation||interest);
22   }
23
24   private static boolean random() {
25     return Math.random() < 0.5;
26   }
27 }
```

Figure 4: Variability encoding of the feature model given in Figure 1.

MonKeY.

The theorem prover KeY can be used to prove that a Java program fulfills its JML specification [9]. KeY comes with a graphical user interface that allows a user to select and prove that each method fulfills its method contract including specified class invariants. The user is able to inspect proof obligations that KeY generates for a given Java program with JML specification. To verify a proof obligation, the user can apply inference rules to split the proof obligation or to transform it into assumptions, whereas KeY checks that each rule application is admissible. Furthermore, KeY uses heuristics to find proofs automatically. But, due to the undecidability of the underlying verification problem, we cannot conclude that there is no proof if KeY is not able to find one. Thus, the interactive mode is helpful when KeY is not able to complete a proof automatically.

The workflow in KeY is cumbersome, if a user wants to verify the correctness of the overall program. The reason is that while many proof obligations can be proved automatically using KeY, the user needs to select every proof obligation manually. To improve usability, we implemented the tool MonKeY as a batch-mode extension for KeY.[3] MonKeY gets a Java program with JML specification as input and uses KeY to try to prove every proof obligation automatically. The output of MonKeY is a table containing all proof obligations stating whether they are proved or not, and if proved, the time needed for proving and the proof complexity in terms of nodes and branches. MonKeY minimizes the user interactions required to prove the correctness of the overall program assuming that proofs can be found automatically. If MonKeY fails to complete a proof, the user can interactively complete the proof using KeY.

[3]`http://www.key-project.org/download/\#monkey`

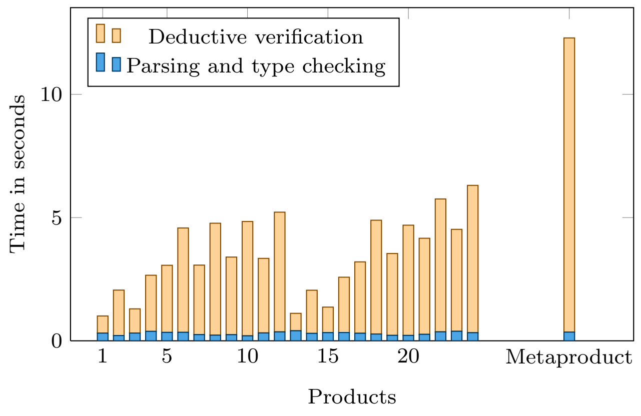

Figure 5: Time needed to verify each product separately (left bars), and the time needed to verify the metaproduct (right bar).

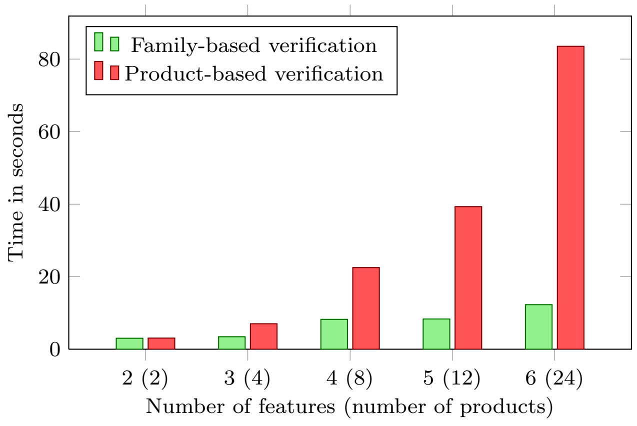

Figure 6: Time needed for family-based verification compared to product-based verification.

Verifying the Bank Account Product Line.

The feature diagram given in Figure 1 describes the bank account product line that we verified in our case study. We implemented six feature modules (excerpts shown in Figure 2) that can be used to generate 24 different products with corresponding specifications. We transformed all feature modules, their specifications, and the feature model into a single metaproduct. Then, we verified the metaproduct using MonKeY. The verification was time-consuming as our initial feature specifications and feature modules contained several errors. Finally after fixing all errors, we were able to prove all contracts automatically.[4]

After completely verifying the metaproduct, we generated and verified each product for comparison. All products have been verified fully automatically. In particular, we did not find any errors in specifications or implementations that the verification of the metaproduct missed (i.e., the metaproduct generation is correct). For the metaproduct and all products, we measured the time needed for automatic verification and the complexity of the resulting proofs.

Verification Time.

In Figure 5, we show the time needed to verify all 24 products. We measured all values using MonKeY and distinguish between the time needed (a) for parsing and type checking the JML-annotated Java program and (b) for automatically proving all contracts. We used a notebook with Intel Core i7-2620M 2.70 GHz processor and 8 GB RAM. We computed average values on ten runs to avoid computation bias. The verification time per product ranges between 1 s and 6.3 s. Overall, verifying all products separately took 83.5 s. In contrast, verifying our metaproduct took only 12.3 s. Thus, our approach saves more than 85 % of computation time for the bank account product line.

In Figure 6, we compare the overall effort of family-based verification with that of product-based verification when the number of features grows. To assess how the effort changes when adding new features, we measured subsets of our bank account product line. We generated and measured all products and the metaproduct for the features *BankAccount* and *DailyLimit*. Then, we iteratively added the remaining features in the following order and repeated our measurements: *CreditWorthiness*, *Interest*, *InterestEstimation*, *Overdraft*. With the product-based approach, the verification time almost doubles with every new feature and thus induces an exponential effort in the number of features. In contrast, the effort for our family-based approach appears to be rather linear in the number of features. Hence, for large software product lines, our approach may save even more than 85 % of the verification effort.

[4]We used the following parameters for verification that MonKeY passes to KeY: `method_treatment=expand`, `dependency_contracts=on`, `query_treatment=on`, and `arithmetic_treatment=defops`.

Proof Complexity.

In Figure 7, we illustrate the complexity of the proof obligations in terms of proof steps (called nodes in KeY) and proof branches. The number of proof steps represents the number of inference rules applied to prove a certain proof obligation. The number of branches indicates how often a proof obligation was split into more than one other proof obligation (a.k.a. case distinction). The most complex proof for the metaproduct took 3.4 s to prove and incorporates 1350 proof steps and 26 branches. The most complex proofs for the products took up to 1.4 s each and incorporates up to 607 proof steps and 20 branches. So, the number of proof steps for the metaproduct is twice as high as for the most complex product, while the number of branches is about the same as for the most complex product.

6. DISCUSSION

In our approach, we use all feature modules as input to generate a metaproduct, which is then used for verification. The problem with verifying generated code is that the user may need to understand the generated code. This is not necessary for automatic theorem proving, but when user interaction is required. This is a limitation of our approach, but this limitation also applies to the standard product-based approach, in which every generated software product is verified separately. It would be useful if theorem provers could be used directly for verifying feature modules, but this is not possible with current theorem-prover technology. We tried

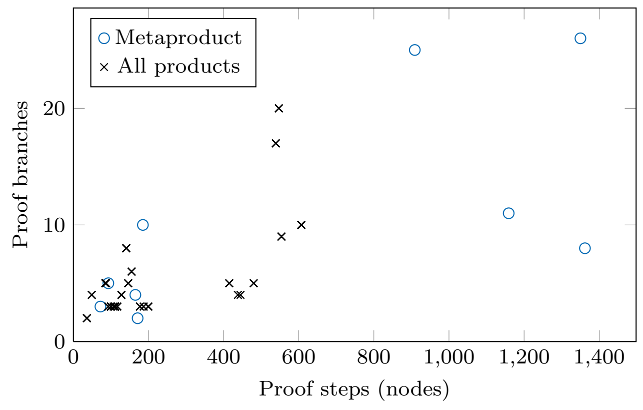

Figure 7: Proof complexity of proof obligations for products and the metaproduct.

to alleviate this limitation by making the transformation into a metaproduct as easy and intuitive as possible.

A general problem with variability encoding for verification of product lines is that verification problems can become large. Basically, it may happen that the verification of each product is possible within certain memory restrictions, but the verification of the metaproduct is impossible. One advantage of design by contract in this respect is that we can actually verify a program by proving that every method fulfills its method contract including specified class invariants. Similarly, with our approach, the verification of a software product line consists of many small proof obligations for every method. But still, the proof for a certain method in the presence of variability is usually harder than verifying the method of a single product. While in the worst case a method may be refined by every feature resulting in complex proof obligations, our experience is that such methods are rare [35]. First, most methods are only introduced and not refined at all [35]. Second, if a method is refined, then there are usually just a small number of refinements [4].

Similarly, method contracts for the metaproduct can get complex due to the transformation. But again, the situation that all features refine the very same method is rare [4, 35]. Furthermore, not every method refinement does actually refine the contract [35]. While we cannot generalize the results from our case study to any software product line, our experiences suggest that the complexity of method contracts and the resulting proof obligations is manageable.

The theorem prover KeY and also many other JML-based verification tools introduce some restrictions on the programs that can be verified. The reason is that all used classes must be specified in JML, which is a problem when using classes from the Java standard library. So far, only parts of the Java standard library have been specified [11]. But, there is better support for Java Card programs, since the Java Card library is almost fully specified [11]. In our case study, we had to provide the method signature for `java.io.Math.random()` to make it available in KeY.

Alternative features according to the feature model may have contradictory specifications in terms of contracts or invariants. Our approach to generate the metaproduct can actually handle contradictory specifications. Basically, this is achieved by transforming each specification into an implication based on the selection of the according feature (see Section 4). Thus, even if two specifications contradict each other, the generated specifications do not contradict each other when the according features are alternative according to the feature model. Hence, the specifications of alternative features can also be proved using our approach.

Our family-based approach of deductive verification is not designed for type checking of software product lines. KeY is able to detect type errors such as dangling method references in Java programs as well as in JML specifications. By checking our metaproduct in KeY, we are not able to detect all type errors, because all methods contained in any feature module are contained in the metaproduct. Thus, our approach should be combined with type checking of the software product line. Fortunately, type errors can be efficiently detected for software product lines [2, 21].

7. RELATED WORK

Variability Encoding for Model Checking.

Variability encoding has been proposed for family-based model checking of software product lines [28, 6]. Post and Sinz use variability encoding (configuration lifting) to verify Linux device drivers implemented with C and preprocessor directives using the bounded model checker CBMC [28]. Similarly, Apel et al. use variablity encoding for feature modules written in C to apply the symbolic model checker CPAchecker [6]. Contrary to this work, they verify programs with a single main method as the sole program entry point. We showed how to encode the feature model for Java programs with multiple main methods or even Java libraries.

JML for Feature-Oriented Programming.

In previous work, we investigated the formal specification of feature-oriented programs using contracts defined in JML [35]. We presented and discussed five specification approaches for feature-oriented programming and compared them regarding strictness, expressiveness, complexity, and their ability to avoid specification clones. By means of case studies, we found that method refinements often require the refinement of contracts, and that the refinement of invariants can and should be avoided. Furthermore, we proposed proof composition for deductive verification of feature-oriented programs [34]. Contrary to this work, (a) we aimed at interactive theorem proving using the proof assistant Coq, and (b) we focused on composing proof scripts rather then making the verification tool variability-aware. Proof composition is not a family-based approach, as proof scripts are written for every feature and must be checked using the verification tool for every product. Finally, we used JML for the detection of feature interactions pursuing a product-based approach based on ESC/Java2 [30].

Contracts for Delta-Oriented Programming.

Contracts have also been discussed for delta-oriented programming [10, 17]. Delta-oriented programming can be seen as an extension of feature-oriented programming, in which feature modules (delta modules) can also remove classes, methods, and fields. Bruns et al. show how to verify each product separately by reusing verification effort of previously verified products using delta-oriented slicing [10]. But still, every product needs to be generated and verified, even if the overall verification effort is reduced. Contrary to our approach, verification still involves redundant computations, because only one verified product is used for the verification of each product. Hähnle and Schaefer present a deductive verification approach relying on the Liskov principle [17]. They used the abstract behavioral specification language to specify delta modules. Their compositional verification principle enables the verification of each delta in isolation and the conformance checking of all delta modules using a family-based approach.

Contracts for Aspect-Oriented Programming.

In the last decade, design by contract has been applied to aspect-oriented programming [23, 36, 26, 22, 1]. The aspect-oriented around advice is similar to feature-oriented method refinement, thus aspect-oriented programming can be seen as a superset of feature-oriented programming [5]. The community working with aspect-oriented programming focuses on modularity rather then variability. Thus, the absence of aspects is usually not considered, while optional features in feature-oriented programming are essential for variability in software product lines. Using these approaches for the verification of software product lines usually requires to generate and verify each combination of aspects separately in a product-based fashion. Contrary, our main goal is to avoid redundant verification and thus to save verification effort.

Contracts for Context-Oriented Programming.

Context-oriented programming is used by Hirschfeld et al. to implement dynamic contract layers [18]. Context-oriented programming can be used to implement variability, whereas the feature selection (context) can be changed at runtime. Context-oriented programming is similar to feature-oriented programming as method implementations depend on a certain context. In feature-oriented programming, the feature selection is fix and must be known at compile-time already. The idea of dynamic contract layers is that runtime assertion checking is extended such that contracts can be activated or deactivated during runtime. They assign contracts to contexts, which is similar to our definition of contracts for each feature, but they do not tackle variability.

8. CONCLUSION

The success of software product-line engineering depends not only on efficient implementation techniques, but also on efficient analysis techniques. We present a family-based approach to verify software product lines using theorem proving. Our approach avoids redundant verification tasks, by making the verification tool aware of the variability.

We presented how to apply variability encoding to feature modules written in Java and their corresponding JML specifications. We showed how to generate a metaproduct that simulates the behavior of all products and a corresponding metaspecification describing the intended behavior of all products. We evaluated our approach by means of a case study using the theorem prover KeY. We compared our family-based approach with a product-based approach verifying each product separately and found that verification time can be reduced by more than 85 %.

In future work, we want to generalize our approach to product lines that are not type uniform or that may contain name shadowing, and to further JML language constructs. Furthermore, we will formalize the generation of metaproduct and metaspecification to prove soundness and completeness of our approach. Finally, we plan to investigate other specification approaches for feature modules and different techniques for metaproduct generation.

9. ACKNOWLEDGMENTS

We thank Richard Bubel for comments on earlier drafts and for assisting us with KeY. We gratefully acknowledge our beneficial discussion with Erik Ernst. Schaefer's work is supported by the German Research Foundation (DFG – SCHA 1635/2-1). Apel's work is supported by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13). Hentschel's work is supported by the European Union (EU – FP7-231620 HATS).

10. REFERENCES

[1] S. Agostinho, A. Moreira, and P. Guerreiro. Contracts for Aspect-Oriented Design. In *Proc. Workshop Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, pages 1:1–1:6, New York, NY, USA, 2008. ACM.

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

[3] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 2012. To appear.

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

[5] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering (TSE)*, 34(2):162–180, 2008.

[6] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375, Washington, DC, USA, 2011. IEEE.

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

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

[9] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Heidelberg, New York, London, 2007.

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

[11] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[12] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 321–330, New York, NY, USA, 2011. ACM.

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

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

[15] A. Fantechi and S. Gnesi. Formal Modeling for Product Families Engineering. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 193–202, Washington, DC, USA, 2008. IEEE.

[16] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proc. IFIP Int'l Conf. Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 113–131, Berlin, Heidelberg, New York, London, 2008. Springer.

[17] R. Hähnle and I. Schaefer. A Liskov Principle for Delta-oriented Programming. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 190–207, Karlsruhe, Germany, 2011. Technical Report 2011-26, Department of Informatics, Karlsruhe Institute of Technology.

[18] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic Contract Layers. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 2169–2175, New York, NY, USA, 2010. ACM.

[19] J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

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

[21] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *Trans. Software Engineering and Methodology (TOSEM)*, 21(3), 2012. To appear.

[22] S. Katz. Aspect Categories and Classes of Temporal Properties. In *Trans. Aspect-Oriented Software Development*, pages 106–134, Berlin, Heidelberg, New York, London, 2006. Springer.

[23] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect Composition Applying the Design by Contract Principle. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE)*, pages 57–69, Berlin, Heidelberg, New York, London, 2001. Springer.

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

[25] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[26] D. H. Lorenz and T. Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *Computing Research Repository (CoRR)*, abs/cs/0501070, 2005.

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

[28] H. Post and C. Sinz. Configuration Lifting: Software Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350, Washington, DC, USA, 2008. IEEE.

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

[30] W. Scholz, T. Thüm, S. Apel, and C. Lengauer. 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, 2011. ACM.

[31] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 47–60, New York, NY, USA, 2011. ACM.

[32] T. Thüm. Verification of Software Product Lines Using Contracts. In *Magdeburger Informatik Tage (MIT)*, 2012. To appear.

[33] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, 2012.

[34] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, DC, USA, 2011. IEEE.

[35] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 255–269, Berlin, Heidelberg, New York, London, 2012. Springer.

[36] J. Zhao and M. C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 150–165, Berlin, Heidelberg, New York, London, 2003. Springer.