Variability Hiding in Contracts for Dependent Software Product Lines

Thomas Thüm, Tim Winkelmann
TU Braunschweig
Germany

Reimar Schröter
University of Magdeburg
Germany

Martin Hentschel, Stefan Krüger
TU Darmstadt
Germany

ABSTRACT

Software product lines are used to efficiently develop and verify similar software products. While they focus on reuse of artifacts between products, a product line may also be reused itself in other product lines. A challenge with such dependent product lines is evolution; every change in a product line may influence all dependent product lines. With variability hiding, we aim to hide certain features and their artifacts in dependent product lines. In prior work, we focused on feature models and implementation artifacts. We build on this by discussing how variability hiding can be extended to specifications in terms of method contracts. We illustrate variability hiding in contracts by means of a running example and share our insights with preliminary experiments on the benefits for formal verification. In particular, we find that not every change in a certain product line requires a re-verification of other dependent product lines.

CCS Concepts

•Software and its engineering → Software product lines; Formal software verification; Feature interaction; Abstraction, modeling and modularity;

Keywords

Multi product line, deductive verification, method contracts

1. INTRODUCTION

Software product-line engineering is a paradigm to efficiently develop similar software systems [2]. A software product line refers to a set of software products that share features and software artifacts. Typically, each software product is generated automatically given a selection of features (i.e., configuration), whereas valid combinations of features are documented in a feature model [3]. The generation causes reduced development effort and at the same time, sharing artifacts across software products reduces the verification effort [25]. A promising strategy to verify product lines with existing verifiers is variability encoding [29]; the compile-time variability of a product line is translated into runtime variability for verification purposes.

The growing adoption of product lines naturally leads to the challenge of reusing product lines themselves. A product line facilitates reuse between products, whereas, for a certain product line, we may want to reuse another product line or parts thereof. A characteristic of such dependent product lines (aka. multi product lines [13]) is that the configuration of one product line influences the configuration of another product line. In particular, it is often not sufficient to just reuse one product of a product line for all products of another. Whereas dependencies between features can be modeled with dependent feature models [14, 23], the verification of feature's implementation is challenging due to evolution: Each change of an artifact in one product line can potentially break any product of other product lines. Even worse, product-line evolution is more frequent than for single systems, as the evolution of one artifact may lead to the evolution of multiple products.

With variability hiding, we propose to hide unnecessary details of one product line for other product lines. Thus, in principle, variability hiding for product lines is similar to information hiding in programming languages, but the variability itself accompanies several new challenges. In prior work, we proposed to face the challenges by means of interfaces on multiple abstraction levels [21]. Whereas we investigated feature-model interfaces to hide features and their dependencies in feature models [20] and feature-context interfaces to hide implementation details based on the reuse context [22], this work facilitates the efficient verification of dependent product lines under evolution. In particular, we make the following contributions:

- Using feature-oriented contracts for feature-module specification [24], we discuss three strategies to hide features.
- We evaluate these strategies when verifying two dependent product lines, a bank account and a bank application, whereas variability encoding and verification is performed with FeatureIDE and KeY.
- We share our insights with applying variability hiding to contracts and discuss open challenges.

2. BACKGROUND

By means of a running example, we briefly introduce feature models, feature modules, feature-oriented contracts, as well as variability encoding, all required for our discussions.

2.1 Feature Model of Our Running Example

Typically, the features of a product line are not all independent of each other. Thus, using feature models, we can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VaMoS '16, January 27-29, 2016, Salvador, Brazil
© 2016 ACM. ISBN 978-1-4503-4019-9/16/01...$15.00
DOI: http://dx.doi.org/10.1145/2866614.2866628

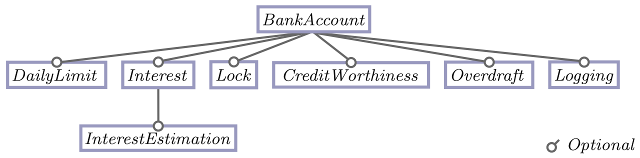

Figure 1: Features in product line *BankAccount*

specify their valid combinations. A feature model (aka. feature diagram) consists of a hierarchical structure and cross-tree constraints [3, 2]. The selection of a feature always requires the selection of its parent feature, whereas the root feature is always selected. Although there are several types of feature groups, for the sake of our example it is sufficient to know that optional features may or may not be selected when their parent is selected. Cross-tree constraints can be based on arbitrary propositional formulas [3].

In Figure 1, we show the feature model of a bank-account software (*BankAccount*), which provides specialized products to manage a bank account. Feature *BankAccount* is the root feature and provides basic storing functionality for all products. There are optional features to define a maximal daily withdrawal (*DailyLimit*), to calculate interests for the account (*Interest*), to predict the expected interest (*InterestEstimation*), to withdraw more money than available up to a certain limit (*Overdraft*), to calculate whether the customer may get a credit (*CreditWorthiness*), to lock the account (*Lock*), and to log all changes (*Logging*). All features are independent of each other except that *BankAccount* is always selected and *InterestEstimation* requires *Interest*.

For verification purposes, feature models are often translated into propositional formulas, in which each feature is mapped to a boolean variable [6]. For our running example, we can automatically derive the formula $BankAccount \wedge (InterestEstimation \Rightarrow Interest)$.

2.2 Feature Modules

We illustrate our approach with feature-oriented programming, as most tool support for our approach is already available [27]. With feature-oriented programming, products are derived automatically by composing modules for each selected feature [4, 19]. Software artifacts concerning a feature are encapsulated in a feature module. In particular, a feature module may introduce new classes or extend existing classes by adding or refining methods and fields.

In Figure 2, we illustrate feature modules of product line *BankAccount*, whereas feature modules are represented by rows and classes by columns. In our example is just one class `Account` being introduced in feature module *BankAccount* and refined by all other feature modules. Refinements add new methods, such as method `calculateInterest` in feature *Interest*, and refine existing methods, such as method `update` in feature *DailyLimit* (fields are omitted for brevity).

In Figure 3, we show an excerpt of feature module *BankAccount* introducing class `Account` (ignore comments for now). The class provides a constructor, a field to store and a method to update the current balance. In Figure 4, we show the extension made, if features *Overdraft* and *DailyLimit* are selected. In detail, *Overdraft* overrides method `getOverdraftLimit` and returns `-5000` instead of `0`. Feature *DailyLimit* introduces constant `DAILY_LIMIT` storing the maximum daily withdrawal of an account and `withdraw` to store the current withdrawal of the day. Additionally, the method `update` extends the related method of feature *BankAccount*

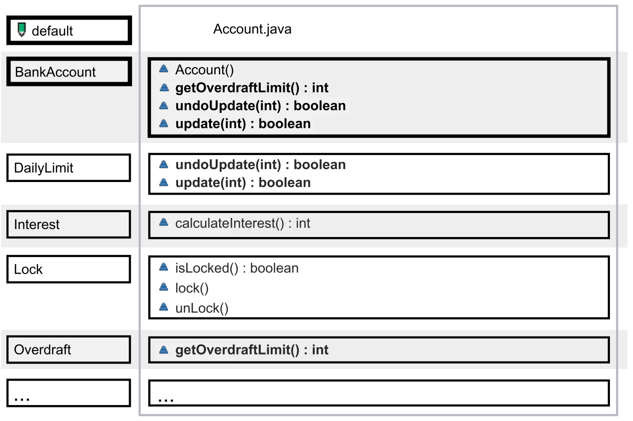

Figure 2: Modules of product line *BankAccount*

using the keyword *original* and makes sure that the withdrawal is updated and does not exceed the limit.

2.3 Feature-Oriented Contracts

Design by contract is a methodology to increase the reliability of software systems with code-level specifications [18]. In particular, a contract is assigned to a method and defines preconditions and postconditions, which are predicates that define valid states before and after method execution [12]. In addition, class invariants allow to specify constraints on the state of objects and classes. For illustration, we use the *Java Modeling Language (JML)* to define contracts and invariants, which can be verified with deductive verification [5]. In deductive verification, Java program and JML specification are translated into logic to prove that the program establishes the specification for all possible inputs [5].

In Figure 3, we exemplify a JML specification. An invariant specifies that field `balance` is not lower than the overdraft limit (cf. Line 2). The constructor's postcondition states that the balance of a newly created account object is zero. For method `update`, we omitted the precondition `true` and the postcondition defines that the balance is only updated, if the return value is true. Keyword `result` refers to the return value, whereas `old` refers to the state of field `balance` prior to method execution. Moreover, keyword `pure` indicates that method `getOverdraftLimit` is side-effect free, which is necessary to call this method within contracts [5].

While JML is designed for Java, we can use it for feature modules similarly [24]. In Figure 4, feature module

```
 1 class Account {
 2   //@ invariant balance >= getOverdraftLimit();
 3   int balance = 0;
 4   //@ ensures balance == 0;
 5   Account() { }
 6   //@ ensures
 7   //@ (!\result ==> balance == \old(balance)) &&
 8   //@ (\result ==> balance == \old(balance) + x);
 9   boolean update(int x) {
10     if (balance + x < getOverdraftLimit())
11       return false;
12     balance += x;
13     return true;
14   }
15   //@ ensures \result == 0;
16   int /*@ pure @*/ getOverdraftLimit() { return 0; }
17 }
```

Figure 3: Feature module *BankAccount*

```
 1 class Account {
 2   //@ ensures \result == -5000;
 3   int /*@ pure @*/ getOverdraftLimit(){ return -5000; }
 4 }
```

```
 5 class Account {
 6   //@ invariant withdraw >= DAILY_LIMIT;
 7   final static int DAILY_LIMIT = -1000;
 8   int withdraw = 0;
 9   //@ ensures \original;
10   //@ ensures !\result ==> withdraw == \old(withdraw);
11   //@ ensures \result ==> withdraw <= \old(withdraw);
12   boolean update(int x) {
13     int newWithdraw = withdraw;
14     if (x < 0)  {
15       newWithdraw += x;
16       if (newWithdraw < DAILY_LIMIT) return false;
17     }
18     if (!original(x)) return false;
19     withdraw = newWithdraw;
20     return true;
21   }
22 }
```

Figure 4: Feature modules *Overdraft* and *DailyLimit*

```
 1 class Account {
 2   //@ invariant bankAccount && (!interestEstimation
 3   //@        || interest);
 4   boolean bankAccount, dailyLimit, interest,
 5           interestEstimation, overdraft, [...];
 6   //@ invariant balance >= getOverdraftLimit();
 7   int balance = 0;
 8   //@ ensures balance == 0;
 9   Account() { }
10   //@ invariant dailylimit ==>
11   //@       withdraw >= DAILY_LIMIT;
12   final static int DAILY_LIMIT = -1000;
13   int withdraw = 0;
14   //@ ensures
15   //@ (!\result ==> balance == \old(balance)) &&
16   //@ (\result ==> balance == \old(balance) + x);
17   //@ ensures dailylimit ==>
18   //@ ((!\result ==> withdraw == \old(withdraw)) &&
19   //@ (\result ==> withdraw <= \old(withdraw)));
20   boolean update(int x) {
21     if (!dailylimit) return update_BankAccount(x);
22     int newWithdraw = withdraw;
23     if (x < 0)  {
24       newWithdraw += x;
25       if (newWithdraw < DAILY_LIMIT) return false;
26     }
27     if (!update_BankAccount(x)) return false;
28     withdraw = newWithdraw;
29     return true;
30   }
31   boolean update_BankAccount(int x) { [...] }
32   //@ ensures !overdraft ==> \result == 0;
33   //@ ensures overdraft ==> \result == -5000;
34   int /*@ pure @*/ getOverdraftLimit(){
35     if (!overdraft) return 0;
36     return -5000;
37   }
38 }
```

Figure 5: Variability encoding for class `Account`

Overdraft specifies a new contract, which completely overrides the contract defined in *BankAccount*. In contrast, feature module *DailyLimit* explicitly reuses the postcondition of method `update` in *BankAccount* with keyword `original`.

2.4 Variability Encoding

A common strategy to reuse existing verification tools for product lines is variability encoding [29]. Using variability encoding, the compile-time variability of feature modules is translated into a single product with runtime variability, called metaproduct. As result, we just have to verify the metaproduct instead of verifying the product for each feature combination in isolation. In particular, variability encoding can also be applied to feature-oriented contracts [27].

In Figure 5, we illustrate the metaproduct for our running example. The metaproduct contains all fields and methods of the feature modules. A boolean variable is introduced for each feature to encode the selection state and to switch between different implementations. Different implementations of a certain method are either inlined as for method `getOverdraftLimit` (Lines 35–36) or the methods are renamed as method `update_BankAccount`. The return value of method `getOverdraftLimit` depends on the value of the new variable `overdraft`. In method `update`, the first statement dispatches between the implementation of feature *DailyLimit* (Lines 22–29) and feature *BankAccount* (Line 31).

Similarly, contracts that do not apply to all products are translated with an implication stating that the predicate only is established if the feature is selected. Examples are the class invariant and postconditions of feature *DailyLimit*. To only verify feature combinations that are valid according to the feature model, we introduce an invariant (cf. Lines 2–3) excluding other combinations (cf. Section 2.1).

3. PROBLEM STATEMENT

Assume we aim to develop a product line *BankApplication* with similar features as product line *BankAccount*. In particular, product line *BankApplication* has an additional feature called *Transaction*, which allows one to transfer money from one account to another in an atomic way. Furthermore, it provides handling for features *DailyLimit* and *Interest*, which are necessary when a new day or year is reached (e.g., interests are added to accounts). Besides this additional behavior, we need functionality of product line *BankAccount*.

For product line *BankAccount*, we specified features and their dependencies in a feature model, implemented each feature in a feature module, specified code members by feature-oriented contracts, and verified altogether by means of variability encoding. Ideally, for product line *BankApplication*, we can reuse existing development *and* verification results of product line *BankAccount*. However, there are numerous challenges involved with such dependent product lines.

The first challenge is how to reuse the feature model of product line *BankAccount*. In principle, we could reuse the complete feature model by means of aggregation [23]. However, then every change of feature model for *BankAccount* requires to analyze the feature model for *BankApplication* again. Even worse, each change in the feature model influences variability encoding and, thus, requires a re-verification of *BankApplication*. In prior work [20], we proposed feature-model interfaces to hide features and their dependencies that are not relevant. In short, given a feature model and a set of features to remove, we can calculate and reuse a feature-model interface, which is a feature model itself that maintains all transitive dependencies.

In Figure 6, we illustrate the feature model for *BankApplication*, whereas the feature-model interface is highlighted using dashed lines. The feature model contains the features *Transaction*, *DailyLimit*, and *Interest* due to their added behavior described above. Feature *Lock* is included additionally, because we have to lock an account during transactions, which is modeled as a cross-tree constraint.

As feature-model interfaces are crucial for our discussions, we give a further example in Figure 7. It is different from our running example only in an artificial requires constraint between feature *DailyLimit* and *InterestEstimation*. With

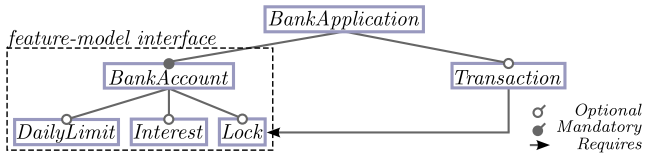

Figure 6: Features in product line *BankApplication*

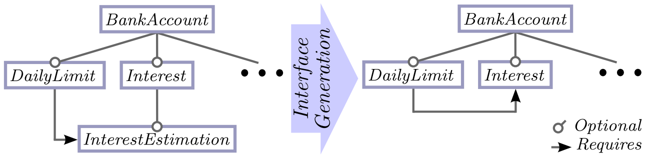

Figure 7: Feature-model interfaces preserve transitive constrains over hidden features

this slight change, a feature-model interface hiding feature *InterestEstimation* contains a transitive constraint between feature *DailyLimit* and *Interest*. That is, the interface does not remove any constraints and describes the same configurations, but with a possibly reduced set of features. A formalization of feature-model interfaces and proofs of those properties are described elsewhere [20].

Assuming a feature-model interface as given, a further challenge is to retrieve accessible implementation artifacts (e.g., methods) for product line *BankApplication*. In Figure 8, we illustrate classes added by product line *BankApplication* and their decomposition into features. Retrieving available artifacts is especially challenging, as it may depend on the context. For example, when implementing method **`transfer`** in feature module *Transaction*, we can safely call method **`lock`** introduced in feature module *Lock* due to the cross-tree constraint *Lock* ⇒ *Transaction* (cf. Figure 6). By contrast, method **`credit`** of feature module *CreditWorthiness* is not accessible. In prior work, we addressed this challenge by introducing feature-context interfaces [22], which automatically present accessible artifacts related to an implementation context. However, it is not clear how to reuse specification and verification effort.

While feature-model interfaces and feature-context interfaces hide features and implementation artifacts, it is still open how to take advantage of hiding during verification. We illustrate the challenge of verifying product line *BankApplication* using method **`transfer`** of class **`Transaction`** in Figure 9. Method **`transfer`** calls method **`update`** of class **`Account`** from the dependent product line *BankAccount*. To verify method **`transfer`**, we can use the contracts of method **`update`** in the metaproduct and, thus, abstract from the actual implementation of method **`update`** with deductive ver-

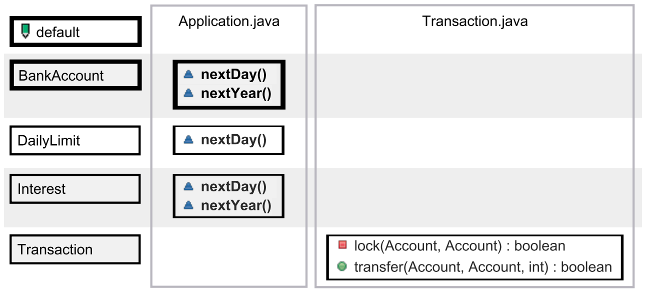

Figure 8: Modules of product line *BankApplication*

```
boolean transfer(Account src,
                      Account dest, int amount) {
  if (!lock(src, dest)) return false;
  try {
    if (amount <= 0) return false;
    if (!src.update(-amount)) return false;
    if (!dest.update(amount)) {
      src.undoUpdate(-amount);
      return false;
    }
    return true;
  } finally { src.unLock(); dest.unLock(); }
}
```

Figure 9: Method `transfer` of class `Transaction`

ification [5]. However, in contrast to our example of Figure 5, the metaproduct is based on all features and, thus, it also contains the contracts of feature *Logging* (cf. Figure 2). Consequently, the metaproduct refers to a feature, which we decided to hide with a feature-model interface. Adding the feature *Logging* to the feature-model interface is not intended, as it reveals details about the implementation and specification of product line *BankAccount*, which are not required for verifying product line *BankApplication*.

4. VARIABILITY HIDING IN CONTRACTS

Feature-model interfaces hide certain features and their dependencies [20], whereas feature-context interfaces hide unavailable implementation artifacts based on feature dependencies [22]. Both have in common to hide some of the variability of dependent product lines to support their evolution. With *variability hiding*, we refer to their common principle of applying information hiding to variability. In this paper, we apply variability hiding to specifications for efficient verification of dependent product lines under evolution. In particular, we extend our previous work on feature-model interfaces and feature-context interfaces by *behavioral product-line interfaces*, which hide features from contracts to ease the verification process.

4.1 Variability Hiding and Potential Benefits

Our goal is to hide all features in contracts that are not in the feature-model interface. For example, with the feature-model interface in Figure 6, contracts can only refer to the features *BankAccount*, *Interest*, *DailyLimit*, and *Lock*. With hiding, we verify a product line for all valid configurations in the feature-model interface, whereas each of them represents one or more valid configurations of the reused product line [20]. An underlying assumption of our following discussions is that any of those configurations can be plugged in. For instance, a configuration can be chosen that is optimal for certain non-functional properties (e.g., footprint). As a consequence, we need to be careful when removing features from contracts, because we somehow need to consider their selection and deselection during verification, without any explicit reference to the feature itself.

The goal of variability hiding in contracts is to save verification effort as illustrated in Figure 10. The verification of our running example without behavioral interfaces requires one to verify *BankAccount* ① and to verify *BankApplication* against *BankAccount* ②. With behavioral interfaces, we verify *BankAccount* ① as above, but verify *BankApplication* against an automatically generated interface ④. Ideally, the interface generation ensures that once *BankAccount* is verified ① it also establishes the interface ③. Furthermore, we assume that given *BankApplication* is verified with the in-

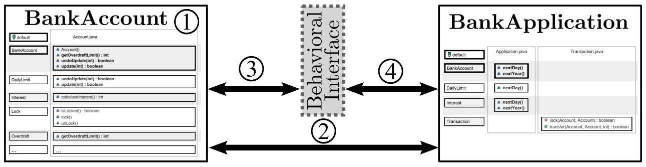

Figure 10: Verification with(out) variability hiding

terface ④, it can also be verified without the interface ②. Proofs for both properties are out of scope and open for future work. Instead, we focus on discussing the benefits for evolution, and evaluating them for our running example.

Besides possibly reduced effort for a single verification run, in principle, variability hiding may reduce the effort under evolution. We distinguish between four different evolution scenarios illustrated using our running example:

1. A change in product line *BankAccount* does *not* require to change the interface: *BankAccount* is re-verified ① and the interface is generated to check for changes.
2. A change in product line *BankAccount* requires to change the interface: *BankAccount* is re-verified ① and the interface is generated to re-verify *BankApplication* ④.
3. A change in product line *BankApplication* does *not* require to change the interface: *BankApplication* is re-verified to the interface ④.
4. A change in product line *BankApplication* requires to change the interface: The interface is generated to re-verify *BankApplication* ④.

Only in one of these four scenarios we need to re-verify both, product line *BankAccount* ① and *BankApplication* ④. In all other cases, we can save verification effort under evolution. However, even in this particular case, hiding some of the variability in contracts may itself provide a benefit.

4.2 Interface Strategies for Variability Hiding

In the following, we investigate strategies to automatically generate behavioral interfaces given a product line with feature-oriented contracts and a feature-model interface. For this process, we rewrite these contracts such that they only depend on features in the feature-model interface. We identified three alternative strategies to remove features, namely false-configuration, true-configuration, and hidden-configuration. While only one of these strategies can be used for a specific feature, we can combine them by applying each strategy to remove some of the features, as each strategy comes with limitations.

False-Configuration. A simple strategy, called false-configuration, is to generate a metaproduct with only features of the feature-model interface. However, the result is semantically equivalent to a deselection of every feature to be removed. Therefore, we identify features in preconditions and postconditions that are not in the feature-model interface, remove the reference and replace the variable in formulas with false. Afterwards, we simplify the resulting precondition and postcondition. Additionally, we can remove any method that is generally prohibited to access by the false feature value in contracts. In our example, this is the case for method `estimatedInterest` if we remove feature *InterestEstimation*, as the precondition (`interestEstimation && (daysLeft >= 0)`) evaluates to false.

Unfortunately, we cannot apply this strategy in all cases. Assume, we would create an interface with feature *InterestEstimation* but without feature *Interest*, then deselecting *Interest* would imply the deselection of *InterestEstimation*, which is a contradiction compared to the dependencies in the interface. In summary, we can only deselect features, whose deselection does not violate the interface. In other words, each configuration in the interface must have at least one corresponding configuration in the product line.

True-Configuration. Analogous to the interface strategy false-configuration and motivated by the previous example, it is also possible to bind variability by selection. In strategy true-configuration, we apply the same transformation principle but replace each occurrence of a feature to remove by `true`. However, for our feature-model interface in Figure 6, the feature *Interest* becomes a mandatory feature, since its child feature *InterestEstimation* is part of every product. Although this strategy is also not applicable in all cases, it could be valuable to remove some features with this strategy.

Hidden-Configuration. While the previous two strategies bind variability by considering features either as selected or deselected, with strategy hidden-configuration, we consider both possible values. As result, the strategy is more complex and we have to handle the removal of preconditions and postconditions differently.

In postconditions, we use a procedure that we proposed earlier to eliminate features from feature models specified as propositional formulas [20, 26]. We combine all ensures-clauses into a single postcondition and duplicate the complete formula. We replace the feature with `true` in the first copy and with `false` in the second copy. Both formulas are then combined using a disjunction and then simplified, if possible. The rationale behind this strategy is that the selection value of the feature can either be `true` or `false`. Consequently, we can rely on the fact that one of these two resulting formulas is fulfilled.

For instance, feature *Overdraft* is not part of the feature-model interface in Figure 6 and, thus, we need to eliminate all occurrences in the metaproduct (cf. Listing 5). The postcondition of method `getOverdraftLimit` is

```
ensures !overdraft || (\result == -5000);
ensures overdraft || (\result == 0);
```

which is transformed into

```
ensures ((!true || \result == -5000) &&
         (true || \result == 0)) ||
        ((!false || \result == -5000) &&
         (false || \result == 0));
```

and then simplified to

```
ensures (\result == -5000) || (\result == 0);
```

being also the intuitive result; if we do not know whether feature *Overdraft* is selected or not, we can only rely on the fact that the return value is either `-5000` or `0`.

In preconditions, we use the same procedure as above, but combine both formulas with a conjunction.[1] A conjunction is used as we have to fulfill the precondition for each possible selection (i.e., `true` and `false`). Otherwise, we could choose a product with a violated precondition. If we need to remove more than one distinct feature value in a certain precondition or postcondition, we iteratively remove every feature with the same procedure.

In principle, the conjunction in precondition may lead to unsatisfiable predicates. This is only problematic, if this particular method is actually called by the reusing product

[1]In contrast, behavioral subtyping is achieved by disjunction in preconditions and conjunction in postconditions [18].

		Variability hiding ③		
	No hiding ①	False	True	Hidden
Nodes	296,961	37,134	242,699	38,460
Time (min)	13.6	1.95	15.9	1.44
Proofs	18	14	18	14
Products	96	8	4	96

Table 1: Verification effort for product line *BankAccount* with(out) variability hiding (cf. Figure 10)

		Variability hiding ④		
	No hiding ②	False	True	Hidden
Nodes	95,292	27,016	53,983	21,393
Time (min)	5.37	0.770	2.20	0.585
Proofs	11	11	11	11
Products	144	12	6	144

Table 2: Verification effort for product line *BankApplication* with(out) variability hiding (cf. Figure 10)

line. Similarly, the disjunction in postconditions could be too weak to prove certain properties of callers. If contracts are indeed not sufficient, we can either decide to add the corresponding feature to the feature-model interface or chose one of the other two strategies.

Summary. With variability hiding, we remove feature references from contracts to reduce the verification effort of dependent product lines. We discussed two strategies to bind variability, namely false- and true-configuration. That is, we effectively verify only a subset of all possible configurations of the reused product line, ruling out some configurations. Consequently, there is no potential to plug-in the best configuration according to non-functional properties. In contrast, hidden-configuration removes features without binding variability. For all three interface strategies, it is unclear to what extent they improve the efficiency.

5. CASE STUDY

To evaluate the feasibility of variability hiding in contracts, we applied it to our running example, a product line we used in prior studies [24, 27]. However, as it was a single product line with all functionality, we had to decompose it into two smaller product lines. We chose a straightforward decomposition along the classes; product line *BankAccount* provides core functionality based on class `Account`, whereas all other classes are defined in product line *BankApplication*.

As decomposition did not break any existing proofs with KeY [5], we started with a verified dependent product line (cf. ① and ② in Figure 10). In KeY, we set method treatment to *contract* rather than inlining to abstract from implementation details when verifying product line *BankApplication*. Otherwise, every change to product line *BankAccount* could potentially break all proofs of *BankApplication*.

We manually applied each interface strategy to all features by potentially restricting the verified products. For plausibility, we verified class `Account` against the interface ③, whereas this verification is not necessary, if we are able to prove that an automatically extracted interface leads to the same verification results as for the original product line (cf. Section 4). As expected, all proofs passed in this case. Finally, we verified all classes of product line *BankApplication* against the interface ④, which is supposed to replace the complete verification ②. Again, KeY verified all methods automatically. We performed all experiments on a notebook with Intel Core i7 (2.4GHz), 8 GB RAM, and Windows 7.

In Table 1, we summarize the verification effort for product line *BankAccount* and its verification against the interface (cf. Figure 2). In detail, we present the verification effort measured in time, nodes, and number of proofs that are necessary to verify. The number of proofs varies as we removed methods with precondition `false`. The number of products varies as false- and true-configuration actually bind variability. As discussed earlier for true-configuration, feature *Interest* effectively becomes a mandatory feature.

As result, we can see that each interface strategy reduces the number of nodes, time, and proofs that are necessary for the verification compared to the verification of the whole product line *BankAccount*. Whereas the verification effort with false- and hidden-configuration noticeably reduce the effort, strategy true-configuration only presents slight improvements during the verification process. Even if our assumption is wrong that these proofs are not necessary, it still seems feasible to perform proofs for interfaces additionally.

In Table 2, we present the verification effort for product line *BankApplication*. We verified class `Application` and `Transaction` either with product line *BankAccount* ② or the behavioral interface ④. Whereas the number of proofs was equal for all scenarios, we get noticeable benefits for number of nodes and time needed for the verification using all interface strategies. Again, strategy true-configuration presents the smallest benefit, while the other two strategies need less than half the time for verification. Especially interesting is that strategy *hidden-configuration* achieves best performance without binding variability: verification time is reduced by 89% and number of nodes by 78%.

As result of our investigation, we conclude that using a behavioral interface can reduce the verification effort for dependent product lines. Overall, we experienced the best performance with strategy hidden-configuration, followed by false- and true-configuration. Strategy true-configuration yields worst results, as it forces the inclusion of all contracts that are introduced by hidden features. Consequently, the complexity of contracts increases and negatively influences the verification effort. It seems that strategy hidden-configuration effectively simplifies contracts, while maintaining all necessary properties in our case study. While this strategy was applicable to all features and all methods in our case study, it is possible that other dependent product lines contain cases in which preconditions are too strong or postconditions are too weak (cf. Section 4). As a fallback, we can still recommend the other strategies or simply to add further features to the feature-model interface (i.e., to remove fewer features from contracts).

6. INSIGHTS AND CHALLENGES

By means of our case study, we convinced ourselves that variability hiding is feasible for contracts and reduces the verification effort. However, we faced several challenges for variability hiding in contracts that we discuss in this section.

Class Invariants. While our previous discussions focused on contracts, classes can also be specified by means of class invariants. Class invariants can significantly reduce the specification effort, as properties that occur in all preconditions and postconditions can be extracted. Furthermore, class invariants are especially helpful for subtyping, as all methods in subclasses have to adhere to the supertype invariants.

The interface strategies false- and true-configuration can directly be applied to invariants. For instance, assume we aim to hide a feature f introducing a class invariant i. In the metaproduct, the class invariant is transformed into $f \Rightarrow i$. Replacing f with false leads to a tautology (i.e., i does not apply), whereas replacing f with true leads to invariant i (i.e., i applies independent of configuration).

In contrast, we do not have an elegant solution for applying strategy hidden-configuration to invariants. In our case study, we had to eliminate each invariant containing features to remove. We eliminated each invariant by adding it to all preconditions and postconditions. For instance, assume a method `m` with precondition ϕ and postcondition ψ, which do not contain feature f for simplicity. Then, adding invariant i to precondition and postcondition results in $\phi \wedge (f \Rightarrow i)$ and $\psi \wedge (f \Rightarrow i)$, respectively. Applying strategy hidden-configuration leads to precondition $\phi \wedge i$ and postcondition ψ. Hence, the resulting contract is not caller-friendly as it relies on the invariant prior execution, but does not establish it again afterwards. For instance, if two methods of the class are executed one after another, the second method's precondition is potentially violated. While no such problems occurred in our case study, a fallback is to add those features to the interface, which, however, would reduce the potential of variability hiding under evolution.

Framing Conditions. For brevity, our previous discussions focused on preconditions and postconditions as being the essence of design by contract. However, we experienced challenges when applying variability hiding to framing conditions, which specify those locations a method is allowed to change. If a feature that we aim to hide introduces a field, we need to remove the field from the interface. Consequently, we need to remove the field also from all framing conditions in which it is mentioned. Although we did not face any problems with removing every occurrence of the field, it is unclear whether this is feasible for other product lines as well. In any case, similar situations occur beyond product lines, when subclasses add new fields and aim to change existing assignable locations of certain methods; the common solution to this are data groups [17], which could be evaluated for these cases in future work.

Alternative Values. While we focused on variability hiding in contracts, we also faced problems with alternative values for fields. For instance, field `DAILY_LIMIT` is introduced with value `-1000` in one feature. Adding an optional feature called *ExtendedDailyLimit* replacing the value with `-2000` leads to `int DAILY_LIMIT = ExtendedDailyLimit ? -2000 : -1000` in the metaproduct. Then, removing feature *ExtendedDailyLimit* from the implementation is not directly possible. One solution is to replace the field access by a getter. In prior work, we experienced similar problems with alternative types when verifying product lines [15, 27]. The solution there is typically to duplicate and rename the field [29]. It is an open question whether we can apply similar solutions for variability hiding.

Java Interfaces and Abstract Classes. With variability hiding, we derive an interface for a given product line and a set of features to be hidden. As realization, we first experimented with Java interfaces on implementation level. However, we soon found out that there are several issues involved when deciding to use Java interfaces or abstract classes for variability hiding. First, assume we already have a good design in a product line, then, it appears wrong to rewrite it into a potentially worse design just to establish variability hiding. In particular, introducing a Java interface for every class requires heavy changes to existing classes of the reused product line (e.g., each class needs to implement the interface then). Second, with variability hiding, we can derive several interfaces for the very same product line customized for a given reuse scenario. That is, by specifying different sets of features to hide, we can derive distinct interfaces. Changing the product line to be aware of all these interfaces appears odd. Finally, as most verification tools establish behavioral subtyping between a class and its interfaces, such an approach may pose severe restrictions on the applicability. All these three arguments also apply to abstract classes, whereas the situation is even worse in view of single inheritance, which would only allow one behavioral interface for each product line.

7. RELATED WORK

We already discussed our prior work on which variability hiding is based on, such as multi-level interfaces [21], feature-model interfaces [20], feature-context interfaces [22], feature-oriented contracts [24], and variability encoding [27, 24, 29].

There are several strategies to scale verification techniques to product lines [25], such as product-based, feature-based, and family-based approaches as well as combinations thereof. In product-based approaches, each product is verified in isolation. In feature-based approaches, each feature's artifacts are verified in isolation, which is typically not sufficient due to feature interactions. In family-based approaches as pursed in this paper, artifacts of all features are analyzed with respect to the feature model.

Others used contracts for deductive verification of product lines. Hähnle and Schaefer propose a feature-family-based approach applying the Liskov principle [10]. The advantage is that features are first verified in isolation, but features' contracts have to adhere to the Liskov principle. Bruns et al. [7] and Hähnle et al. [11] propose optimizations for product-based verification, namely slicing and abstract contracts. Damiani et al. [8] propose feature-product-based verification, whereas features are verified with uninterpreted assertions, which are then checked for each product. With proof composition, we proposed to compose Coq proof scripts of each feature, which are then checked for all products [28]. Most of these approaches require to generate and verify all products, which is infeasible for large product lines and avoided with variability encoding. Furthermore, none of them has been evaluated on dependent product lines.

There are similar approaches for dependent product lines. Kästner et al. propose a variability-aware module system to type check each product line separately [16]. Damiani et al. adapt delta-oriented programming for dependent product lines [9]. While both approaches allow to define feature models for each product line, they do not hide any variability in feature models, implementation, or contracts.

Variability hiding is heavily based on the notion of feature-model interfaces [20], whereas two algorithms to compute them have been proposed. Acher [1] propose to decompose feature models by means of slicing. Thüm et al. [26] eliminate abstract features (i.e., features without mapping to artifacts) to reason about programs instead of valid configurations. Both algorithms remove features from feature models while preserving feature dependencies of the remaining features, which is crucial to the correctness of variability hiding. In addition, strategy hidden-configuration to remove features from contracts is also inspired by those algorithms.

8. CONCLUSION

Evolution challenges the development of dependent product lines. A major problem is that any change in a product line may arbitrarily influence other product lines. With variability hiding, we scale information hiding known from programming languages to the level of product lines. In particular, we apply variability hiding to method contracts in feature-oriented programming. To hide some features to other product lines, we discussed three strategies to remove features from contracts.

The results of applying variability hiding to two dependent product lines indicate that it can drastically reduce the verification effort due to reduced contract complexity. Interestingly, hiding variability is even more efficient than to bind variability. However, further experiments are required, especially to estimate the benefit under typical evolution scenarios. Furthermore, proofs on the relation between verification results with and without variability hiding are needed.

Acknowledgments

We gratefully acknowledge discussions with Richard Bubel and his help in closing all proofs with KeY. We thank all three reviewers for their constructive feedback. This work has been supported by the European Commission (HyVar – H2020-644298), by the German Research Foundation DFG (SCAVES2 – SCHA1635/2-2), and by the Federal Ministry of Education and Research BMBF (NaVaS – 01IS14017B).

9. REFERENCES

[1] M. Acher. *Managing Multiple Feature Models: Foundations, Language, and Applications.* PhD thesis, University of Nice Sophia Antipolis, France, 2011.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20. Springer, 2005.

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

[5] B. Beckert, R. Hähnle, and P. Schmitt. *Verification of Object-Oriented Software: The KeY Approach.* Springer, 2007.

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[7] D. Bruns, V. Klebanov, and I. Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *FoVeOOS*, pages 61–75. Springer, 2011.

[8] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A Transformational Proof System for Delta-Oriented Programming. In *FMSPLE*, pages 53–60. ACM, 2012.

[9] F. Damiani, I. Schaefer, and T. Winkelmann. Delta-Oriented Multi Software Product Lines. In *SPLC*, pages 232–236. ACM, 2014.

[10] R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In *ISOLA*, pages 32–46. Springer, 2012.

[11] R. Hähnle, I. Schaefer, and R. Bubel. Reuse in Software Verification by Abstract Method Calls. In *CADE*, pages 300–314. Springer, 2013.

[12] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *CSUR*, 44(3):16:1–16:58, 2012.

[13] G. Holl, P. Grünbacher, and R. Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *IST*, 54(8):828–852, 2012.

[14] A. Hubaux, T. T. Tun, and P. Heymans. Separation of Concerns in Feature Diagram Languages: A Systematic Survey. *CSUR*, 45(4):51:1–51:23, 2013.

[15] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *TOSEM*, 21(3):14:1–14:39, 2012.

[16] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *OOPSLA*, pages 773–792. ACM, 2012.

[17] K. R. M. Leino. Data Groups: Specifying the Modification of Extended State. In *OOPSLA*, pages 144–153. ACM, 1998.

[18] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., 1st edition, 1988.

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443. Springer, 1997.

[20] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *ICSE*, 2016. To appear.

[21] R. Schröter, N. Siegmund, and T. Thüm. Towards Modular Analysis of Multi Product Lines. In *MultiPLE*, pages 96–99. ACM, 2013.

[22] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *SPLC*, pages 102–111. ACM, 2014.

[23] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated Analysis of Dependent Feature Models. In *VaMoS*, pages 9:1–9:5. ACM, 2013.

[24] T. Thüm. *Product-Line Specification and Verification with Feature-Oriented Contracts.* PhD thesis, University of Magdeburg, Germany, 2015.

[25] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[26] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pages 191–200. IEEE, 2011.

[27] T. Thüm, J. Meinicke, F. Benduhn, M. Hentschel, A. von Rhein, and G. Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *SPLC*, pages 177–186. ACM, 2014.

[28] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *VAST*, pages 270–277. IEEE, 2011.

[29] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. Variability Encoding: From Compile-Time to Load-Time Variability. *JLAMP*, 85(1, Part 2):125–145, 2016.