Using Abstract Contracts for Verifying Evolving Features and Their Interactions

Alexander Knüppel[1], Stefan Krüger[2], Thomas Thüm[3], Richard Bubel[4], Sebastian Krieter[5], Eric Bodden[2], and Ina Schaefer[1]

[1] TU Braunschweig
[2] University of Paderborn
[3] University of Ulm
[4] TU Darmstadt
[5] University of Magdeburg

Abstract. Today, software systems are rarely developed monolithically, but may be composed of numerous individually developed features. Their modularization facilitates independent development and verification. While feature-based strategies to verify features in isolation have existed for years, they cannot address interactions between features. The problem with feature interactions is that they are typically unknown and may involve any subset of the features. Contrary, a family-based verification strategy captures feature interactions, but does not scale well when features evolve frequently. To the best of our knowledge, there currently exists no approach with focus on evolving features that combines both strategies and aims at eliminating their respective drawbacks. To fill this gap, we introduce Fefalution, a feature-family-based verification approach based on *abstract contracts* to verify evolving features *and* their interactions. Fefalution builds partial proofs for each evolving feature and then reuses the resulting partial proofs in verifying feature interactions, yielding a full verification of the complete software system. Moreover, to investigate whether a combination of both strategies is fruitful, we present the first empirical study for the verification of evolving features implemented by means of feature-oriented programming and by comparing Fefalution with another five family-based approaches varying in a set of optimizations. Our results indicate that partial proofs based on abstract contracts exhibit huge reuse potential, but also come with a substantial overhead for smaller evolution scenarios.

1 Introduction

Today's software systems are often developed in terms of features, such as the operating system Linux, the integrated development environment Eclipse, or the web browser Firefox. End-users can choose a subset of these features for a customized product. In such software product lines [7], the number of possible feature combinations typically grows exponentially in the number of features. This kind of variability makes their verification a non-trivial task [40], as individual features change over time due to software evolution, and it is typically impossible

to foresee which product variants need to be generated. Moreover, it is infeasible to generate and verify all feature combinations for even small feature-oriented software projects.

Implementation artifacts of a feature can be modularized into plug-ins, feature modules, or aspects [7]. A major goal of modularity is to reduce complexity and to support large development teams [35]. As a side-effect, modularity allows to verify features to a certain extent in isolation. A major advantage of such *feature-based verification* is that also under evolution one only needs to verify those features that changed [40] and not the complete code base. However, verifying each feature in isolation – known as *feature-based verification* – is generally insufficient, as features interact with each other. For instance, a feature may call methods defined in other features (i.e., syntactical interaction) or it may even rely on the behavior of other features (i.e., semantic interaction) [13]. Any combination of an arbitrary subset of features may lead to a feature interaction. Hence, there is a potentially exponential number of interactions, which are typically unknown a priori.

To resolve the interaction problem, another verification strategy called *family-based verification* applies a single verification to the complete code base (i.e., the whole family of products at once) [10, 15, 22, 24, 38]. Typically, a *metaproduct* is generated by encoding the complete code base including all features into one executable product. While the family-based strategy avoids the verification of every possible feature combination separately, the verification problem is more complex and needs to be solved again whenever one of the features evolves.

Our hypothesis for this work is that a combined approach (i.e., a *feature-family-based strategy*) may be simultaneously robust against evolution of features to reduce re-verification effort and also able to deal with feature interactions. However, testing this hypothesis is a non-trivial task, as there are also numerous *optimizations* (e.g., whether method calls are inlined or abstracted) that may affect the performance and reuse potential of verification strategies in general. To this end, we focus on two open questions in this work, namely (1) *can a feature-family-based strategy reduce the verification effort for evolving features compared to a sole family-based strategy?* and (2) *how do different optimizations affect the verification of evolving features?*

To the best of our knowledge, no adequate feature-family-based approach explicitly addressing the evolution of features currently exists. Therefore, we propose FEFALUTION, a feature-family-based deductive verification approach particularly designed for evolution and for overcoming some limitations by prior work. In a feature-based phase, FEFALUTION builds partial proofs for each evolving feature by generating a *feature stub* containing all artifacts (e.g., classes, methods, and fields) that are referenced from other features. When referring to methods of other features, we use the notion of abstract contracts [11] to avoid that changes in contracts of other features influence the verification result and, thus, require a re-verification. The verification of the feature stubs results in partial proofs.

In a subsequent family-based phase based on *variability encoding* [38], FEFALUTION then reuses the resulting partial proofs in verifying feature interactions. Variability encoding is the process of transforming compile-time into run-time

variability to reuse existing verification tools as-is [43]. We extend prior work on variability encoding for feature-oriented contracts [42] with support for abstract contracts and for partial proofs.

We focus on contract specification with the Java Modeling Language (JML) [32] and verification with KeY [1], whereas our ideas may also be applied to other specification languages and verification tools. As modularization technique, we rely on feature modules specified with feature-oriented contracts due to the available tool support for modularization and composition of contracts [42]. In detail, we make the following contributions.

- A presentation of a feature-family-based deductive verification approach called Fefalution, which combines the generation of feature stubs [31] and abstract contracts [11] with variability encoding [38].
- Tool support for Fefalution based on three existing tools, namely (1) a version of KeY that facilitates the use of abstract contracts, (2) FeatureHouse [4] for composing feature modules and feature-oriented contracts with support for variability encoding, and (3) FeatureIDE [41] for the generation of feature stubs.
- An empirical comparison of a total of six family-based approaches varying in specific characteristics applied to five evolution scenarios of the bank account product line.

2 Background and Running Example

In this section, we introduce the core concepts our work is based on. We briefly discuss JML specifications in general and abstract contracts in particular. As we focus on software product lines, we give a brief introduction to feature modeling, feature composition, and product-line specification.

2.1 Design By Contract

To specify a program's behavior, we follow the *design-by-contract* paradigm [34]. In design-by-contract, the program behavior is typically specified by code annotations that have to be obeyed by method implementations. Method contracts specify *preconditions* that need to be satisfied by callers and *postconditions* that callers can then rely on. Furthermore, class invariants define class-wide properties. A class invariant is established by the class' constructor and serves as an additional precondition and postcondition for each method of the class.

As specification language, we use the Java Modeling Language (JML), which implements design-by-contract for Java programs [33]. In the first listing of Figure 1, we show the contracts of method `update`. The listings in the figure also show the mechanism of composition in feature-oriented programming, which we discuss further below. The keywords **`requires`** (Line 5) and **`ensures`** (Line 6) indicate the pre- and postcondition, respectively. The last line of the contract (Line 7) represents the assignable clause. An assignable clause represents a set of

BankAccount

```
1  class Account {
2    static final int OVERDRAFT_LIMIT = 0;
3    int balance = 0;
4
5  /*@ requires x != 0;
6    @ ensures \result <==> (balance == \old(balance) + x);
7    @ assignable balance; @*/
8    boolean update(int x) {
9      if (balance + x < OVERDRAFT_LIMIT) return false;
10     balance += x;
11     return true;
12   }
13 }
```

DailyLimit

```
14 class Account {
15   static final int DAILY_LIMIT = -1000;
16   int withdraw = 0;
17
18 /*@ requires \original;
19   @ ensures \original;
20   @ assignable withdraw; @*/
21   boolean update(int x) {
22     if ((x < 0 && withdraw + x < DAILY_LIMIT) ||
23       !original(x))
24       return false;
25     withdraw += x;
26     return true;
27    }
28 }
```

BankAccount • DailyLimit

```
29 class Account {
30   static final int OVERDRAFT_LIMIT = 0;
31   int balance = 0;
32   static final int DAILY_LIMIT = -1000;
33   int withdraw = 0;
34
35 /*@ requires x != 0;
36   @ ensures \result <==> (balance == \old(balance) + x);
37   @ assignable balance; @*/
38   boolean update_wrappee_BankAccount (int x)
39   {//Lines 9 - 11}
40
41 /*@ requires x != 0;
42   @ ensures \result <==> (balance == \old(balance) + x);
43   @ assignable balance, withdraw; @*/
44   boolean update(int x) {
45     if ((x < 0 && withdraw + x < DAILY_LIMIT) ||
46       !update_wrappee_BankAccount(x))
47       return false;
48     withdraw += x;
49     return true;
50    }
51 }
```

Fig. 1. Composition of two feature modules of a bank account product line [42]

program locations that the method is permitted to change on return. For method `update`, only field `balance` may change.

Contracts can be used by a theorem prover during the deductive verification process when a method invocation is encountered. Alternatively, method invocations can be treated by inlining the body of the called methods. This has several disadvantages [30], as (a) in presence of dynamic dispatch the verifier has to split the proof produced by the theorem prover into different cases, one for each overwritten method to which the method invocation might be dispatched at runtime, (b) the verification (program analysis) is no longer modular as changes to the implementation of the called method invalidate proofs for all callers, and (c) the verification becomes a closed-world analysis, as the implementation of called methods needs to be accessible in order to guarantee correctness. Furthermore, available implementations for native methods calls, however, are rarely accessible.

These disadvantages can be mitigated by using the contract of an invoked method and following the paradigm of behavioral subtyping. This way, we can avoid to enumerate all possible method implementations and only apply the most general common method contract. Additionally, contracts are usually more stable than implementations as they are only concerned with *what* is computed and not *how* the algorithm is implemented. Hence, we become independent of changes to the implementation of called methods. As inlining may produce an indefinite large call stack, contracts should reduce the proof complexity. However, it is an open question under which circumstances contracts indeed lead to less verification effort [30].

2.2 Abstract Contracts

Hähnle et al. [26] and Bubel et al. [11] propose abstract contracts for the deductive verification of evolving source code. Using abstract contracts in verification, the theorem prover creates partial proofs without relying on the concrete definitions of contracts. These partial proofs are saved and reused to save verification effort during a re-verification. Abstract contracts may reduce the overall verification effort of a program under development [11].

We illustrate the structure of abstract contracts by making the concrete contract of method `update` abstract. We show the method's concrete contract in Lines 5–7 in Figure 1 and the resulting abstract contract in Figure 2. Lines 1–3 of Figure 2 represent the abstract section, in which placeholders for precondition, postcondition, and assignable clause are declared. Lines 4–7 consist of the placeholders' definitions and are called the concrete section. In our example, Line 1 shows the declaration of precondition placeholder `updateR`, which is then defined in Line 4 as `x != 0`. When a verification is performed based on abstract contracts, the theorem prover uses the placeholders instead of the concrete definition in the proving process.

```
1  /*@ requires_abs updateR;
2    @ ensures_abs updateE;
3    @ assignable_abs updateA;
4    @ def updateR = x != 0;
5    @ def updateE = \result <==>
6    @   (balance == \old(balance) + x);
7    @ def updateA = balance; @*/
8    boolean update(int x) {}
```

Fig. 2. Method *update* in role *Account* in feature *BankAccount*

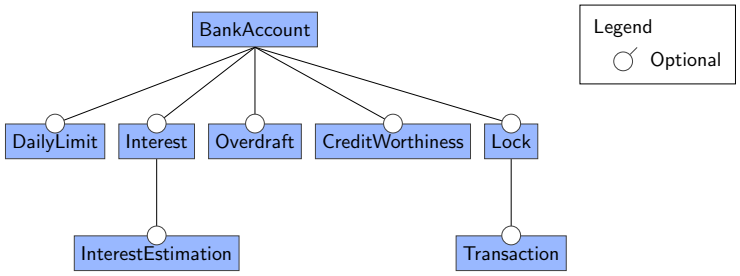

Fig. 3. fig:Feature Model of a Variant of the Bank Account Product Line

2.3 Feature Modeling and Valid Feature Combinations

To assemble a product, feature modules are composed together. However, not all feature combinations are meaningful. For example, it is undesirable to have features in the same product that contain code for specific operating systems. Feature models [14, 29] describe valid combinations of features. The most common way to represent feature models are feature diagrams [29]. In Figure 3, we show the feature diagram that represents our running example, being a software product line implementing a rudimentary BankAccount management system. The root feature *BankAccount* provides a base implementation. All its child features are optional and provide additional functionalities such as maximum daily withdrawal and performing updates of multiple accounts in one atomic transaction. Moreover, some dependencies are imposed on the features. In our running example, feature *InterestEstimation* requires the presence of feature *Interest* and feature *Transaction* requires the presence of feature *Lock*.

2.4 Feature Composition and Specification

In feature-oriented programming, there is a bijective mapping of *features* to so-called *feature modules*. Feature modules encapsulate all the feature's artifacts, such as part of the source code, test cases, or documentation [7]. Importantly, the granularity is not on file level, but different methods and fields of the same class may be *decomposed* into different feature modules with respect to their corresponding feature. Additionally, refined methods in a feature module can be

annotated with a specification (cf. Figure 1). For convenience, we use the terms *feature* and *feature module* interchangeably.

To generate a particular software product p, a set of selectable features $F = \{f_1, \ldots, f_n\}$ is incrementally composed together by a composition operator $\bullet : F \times F \rightarrow F$ [2]:

$$p = f_n \bullet (f_{n-1} \bullet (\cdots \bullet (f_2 \bullet f_1))) \tag{1}$$

For feature modules, the composition is achieved by means of superimposition [3] and for specifications, the composition is achieved by feature-oriented contract composition [42]. Both compositions are not commutative in general [5, 28, 42], such that a developer has to decide in which order features are merged together. In the following, we briefly describe both concepts.

Superimposition [3] is a simple process, where two feature modules represented as trees are recursively composed together by merging their substructures if and only if the parent node is composed and their name and type match. Such a tree structure has to be provided for each programming language individually. For instance, elements of the tree structure for Java programs comprise *packages*, *classes*, *methods*, and *fields* and two classes from composed feature modules are merged if (a) they are named equally and (b) they are part of the same package. In Figure 1, we show an exemplary feature composition with our example product line as it is performed by FeatureHouse [8]. The figure contains the method `update` of features *BankAccount* and *DailyLimit* in the first two listings. In the product line, feature *BankAccount* represents the base feature, with which all other features are composed, meaning that the method implementation of feature *DailyLimit* refines the method implementation of feature *BankAccount*. In the third listing, we show the result of the composition of both methods. The listing contains two methods. In method `update` in feature *DailyLimit* the keyword `original` is used (see Line 23) to call its direct predecessor in feature *BankAccount*. During composition, the keyword is replaced by a call to method `update__wrappee__BankAccount` (see Line 46) defined in feature *BankAccount*.

When a product is generated, its specification needs to be composed as well. While there are several approaches to realize such a contract composition [39, 42], we focus on a particular composition mechanism similar to feature-oriented method refinement on the implementational level, namely *explicit contract refinement* [42]. With explicit contract refinement, refining contracts may refer the original precondition or postcondition in their respective precondition and postcondition by keyword `original` [42]. Let m, m' be two methods specified with preconditions P, P' and postconditions Q, Q', respectively. Then the composition operator for explicit contract refinement is defined as:

$$\{P\}m\{Q\} \bullet \{P'\}m'\{Q'\} = \{P'[\backslash\texttt{original} \setminus P]\}m \bullet m'\{Q'[\backslash\texttt{original} \setminus Q]\} \tag{2}$$

where $P'[\backslash\texttt{original} \setminus P]$ yields the replacement of all placeholders `original` with precondition P in precondition P' ($Q'[\backslash\texttt{original} \setminus Q]$ is defined analogously).

2.5 Family-based verification

The family-based verification that we consider in this work is based on the construction of a *metaproduct*. A metaproduct combines all features of a product line into a single software product by means of *variability encoding* [43]. Essentially, compile-time variability (i.e., the selection of composing features) is transformed into run-time variability (i.e., branching conditions). A boolean class variable is created for each feature that indicates whether a feature is selected (`true`) or not (`false`). These feature variables are then used in implementations and specifications to simulate different feature selections at run-time. Verification tools are configured to treat feature variables as uninitialized to consider all possible combinations.

However, relying only on a family-based strategy when also considering the evolution of product lines has some drawbacks, as all proofs of the former metaproduct may become invalid. For instance, when the feature model has changed, new combinations emerge that were not considered during the last verification. Moreover, a newly added feature might interact with other features. Figure 4 shows how such an interaction is established through method refinements for method `update` in the bank account product line when adding a new feature *Logging*. Method `update` is defined in feature *BankAccount* and refined in features *Logging* and *DailyLimit*. In the metaproduct, all refinements are connected using variability encoding. That is, an if-condition checking whether the respective feature is selected is added to each of the refinements. Since feature *Logging* introduced a new refinement, all `update` methods and all methods that call method `update` need to be re-verified. Therefore, one would need to re-generate the metaproduct, so that new feature *Logging* is included, and verify the new metaproduct again.

Although there is a feature-family-based approach by Hähnle and Schaefer [25] that facilitates proof reuse, it requires a refinement to have more specialized contracts than the method it refines. Feature *Logging*, however, introduces new fields and its methods use them in their contracts. Thus, their approach cannot be applied to the bank account product line under the given evolution scenario.

3 Applying Feature-Family-Based Verification under Evolution

In this section, we introduce our reference approach following a feature-family-based verification strategy called FEFALUTION.

3.1 Overview

Our main research goal is to evaluate whether a feature-family-based verification approach may outperform existing family-based approaches for the verification of product lines under evolution. As to the best of our knowledge no such approach currently exists, we propose a novel two-phased approach following the feature-family-based strategy. Figure 5 presents an overview of FEFALUTION, which is

```
1  public  class Account {
2
3      boolean update(int x) {
4          if (!FM.FeatureModel.Logging) {
5              update_wrappee_DailyLimit(x);
6          } else {
7           //Body of method update from Feature Logging
8          }
9      }
10
11     boolean update_wrappee_DailyLimit(int x) {
12         if (!FM.FeatureModel.DailyLimit) {
13             update_wrappee_BankAccount(x);
14         } else {
15             //Body of method update from Feature DailyLimit
16         }
17     }
18
19     boolean update_wrappee_BankAccount(int x) {
20         //Body of method update from Feature BankAccount
21     }
22 }
```

Fig. 4. Variability Encoding For Method *update*

divided into a *feature-based verification* phase and a subsequent *family-based* verification phase.

The feature-based verification phase mainly consists of two steps. First, a set of feature modules is transformed into a set of *feature stubs* (1). The reason is that feature modules do not typically constitute valid Java programs. Therefore, compiling or verifying them produces type errors. To lift a feature module to a valid Java program, we adopt and extend the concept of *feature stubs* as proposed by Kolesnikov et al. [31] to enable feature-based type checking. Feature stubs extend feature modules with additional (dummy) source code, such that all type and compilation errors are resolved. After the feature-stub generation, the feature stub contains two kinds of methods. For an easier distinction, we refer to methods that originally belonged to the feature module as *domain methods* and we refer to methods created to match calls to methods outside the feature as *method prototypes*. In addition, our realization of feature stubs also resolves dependencies on the level of contracts and adds *pure abstract contracts* to method prototypes to enable a contract-based formal verification.

Second, each method in a feature stub is verified and a corresponding proof is produced (2). For many methods, only partial (or *incomplete*) proofs exist, as they may invoke methods that are only visible in other features and whose concrete definition is therefore unknown at this stage. An additional optimization applies the feature-based verification only if a proof does not already exist in the previous version (e.g., as prevalent for any method of the initial version). Otherwise, the partial proofs of the former version are considered and a proof replay mechanism is applied to also minimize verification effort in the presence of implementation changes.

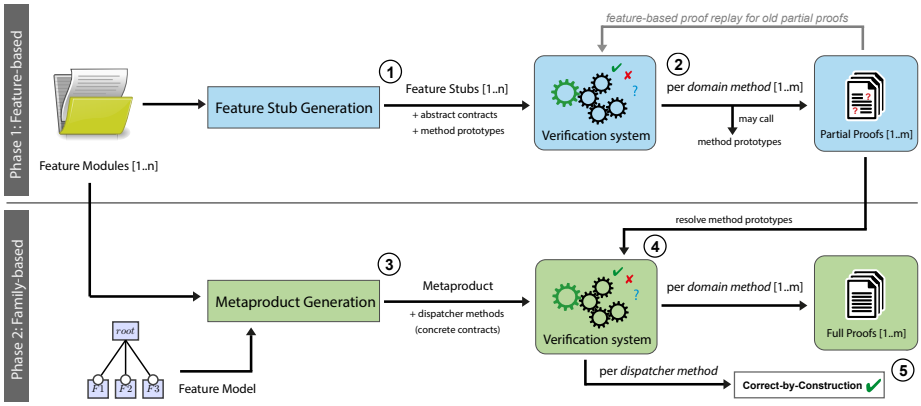

Fig. 5. Overview of FEFALUTION

In the family-based verification phase, FEFALUTION tries to complete all partial proofs. First, the feature modules together with the feature model are used as input to generate the metaproduct (i.e., a single software product that represents the complete product line) (3). Second, FEFALUTION finalizes all partial proofs by replacing the incomplete method invocations (i.e., method prototypes) with the concrete instances and replaying these proof artifacts on the corresponding domain method of the metaproduct (4).

An additional optimization is the generation of *dispatcher methods* to further increase the reuse potential. As illustrated in Figure 4 and proposed by Thüm et al. [38], an if-statement is added for each *domain method* to check whether the corresponding feature is deselected. In this case, the previous method along the feature composition order is called, and introducing additional methods for the variability encoding is avoided. Yet, using only one method is impractical for our approach, as it would be more difficult to reuse partial proofs from the feature-based phase. Therefore, we also introduce *dispatcher methods* in the metaproduct, adopting parts of the variability encoding from Apel et al. [9]. Figure 7 shows an example of a dispatcher method in Line 35. Dispatcher methods serve as a connecting link between two domain methods, and we consider them to be correct-by-construction (5), as their contract only represents the case distinction between both dispatched methods. This way, the corresponding proof obligations become trivially true. In Algorithm 1, we draft our main algorithm, where we denote by DM a set of domain methods and P_{part} a set of partial proofs. We use the prime symbol (e.g., DM') to refer to the respective representation in the previous version.

3.2 From Feature Stubs to Partial Proofs

For each feature-module dependency, FEFALUTION generates a stub for the missing element. For example, consider Figure 6. Method `transfer` of class

Algorithm 1 FEFALUTION($\mathbb{F}$, $\mathbb{F}'$, $\mathcal{FM}$, ∂_{old}), where $\mathbb{F}$ is the set of feature modules, $\mathbb{F}'$ is the set of feature modules from the previous version, $\mathcal{FM}$ is the feature model, and $\partial_{old} : DM' \to P'_{part}$ is a function mapping domain methods $m' \in DM'$ from the previous version to their respective partial proof $p' \in P'_{part}$.

1: Create for each feature module $fm \in \mathbb{F}$ a corresponding feature stub by applying

$$s_{fm} = \text{GENERATEFEATURESTUB}(fm, \mathbb{F}, \mathcal{FM}) \tag{3}$$

and define by $\mathbb{S}_{\mathbb{F}}$ the set of all feature stubs for feature modules $\mathbb{F}$.

2: Compute set $\mathbb{S}_{\mathbb{F}'}$ of all feature stubs for the past feature modules $\mathbb{F}'$. Define $\Delta(\mathbb{S}_{\mathbb{F}}, \mathbb{S}_{\mathbb{F}'}) \subseteq DM$ as the set of *domain methods* that changed.

3: Define $\partial : DM \to P_{part}$ as a function that maps domain methods $m \in DM$ from the current version to their respective partial proof $p \in P_{part}$ by the following case distinction:

a. for each new domain method $m \in DM \setminus DM'$:

$$\partial(m) := \text{FEATUREBASEDVERIFICATION}(m) \tag{4}$$

b. for each domain method $m \in \Delta(\mathbb{S}_{\mathbb{F}}, \mathbb{S}_{\mathbb{F}'})$:

$$\partial(m) := \text{FEATUREBASEDPROOFREPLAY}(m, \partial_{old}(m)) \tag{5}$$

c. for each domain method $m \in DM \cap DM' \wedge m \notin \Delta(\mathbb{S}_{\mathbb{F}}, \mathbb{S}_{\mathbb{F}'})$:

$$\partial(m) := \partial_{old}(m). \tag{6}$$

4: Compute the metaproduct mp based on the current set of future modules $\mathbb{F}$ and feature model $\mathcal{FM}$:

$$mp = \text{COMPUTEMETAPRODUCT}(\mathbb{F}, \mathcal{FM}). \tag{7}$$

5: Define P^{ad}_{part} as the set of adapted partial proofs with respect to metaproduct mp and $\partial^{ad} : DM \to P^{ad}_{part}$ as the mapping function.

6: For all domain methods $m \in DM$, apply

$$p_{full} = \text{FAMILYBASEDVERIFICATION}(m, \partial^{ad}) \tag{8}$$

to obtain the full proof for domain method m by completing the abstract segments in $\partial^{ad}(m)$.

Transaction contains a call to method `update`, which is part of class *Account*. Feature *Transaction*, however, neither contains the class nor the method. Thus, we generate the class and a method prototype as part of the feature stub for feature *Transaction* as shown in Figure 6, which resolves an otherwise emerging type error. Other examples are references that access either a field or type of outside the feature as a feature-module dependency as well.

A special case of a feature-module dependency is keyword `original`. Methods may be refined multiple times by different features, which results in a *refinement chain*. The previous refinement of a method in a refinement chain can be accessed via keyword `original`, which is replaced by a concrete instance of the respective method at compile time. Method `update`, which we show in the feature *DailyLimit* of Figure 1, uses `original` to refer to the previous refinement. In the feature stub generation process, the keyword is replaced with a call to

```
1  class Transaction {
2      public boolean transfer(Account source,
3          Account destination, int amount) {
4          [...]
5          if (!source.update(-amount))
6              return false;
7          if (!destination.update(amount)) {
8              source.undoUpdate(-amount);
9              return false;
10         }
11         [...]
12     }
13 }
```

```
14 class Account {
15    /* method prototype */
16   /*@ requires_abs updateR;
17     @ ensures_abs updateE;
18     @ assignable_abs updateA; @*/
19     boolean update(int x) { return true;}
20
21    /* method prototype */
22   /*@ requires_abs updateR;
23     @ ensures_abs updateE;
24     @ assignable_abs updateA; @*/
25     boolean undoUpdate(int x) { return true;}
26 }
```

Fig. 6. Classes *Transaction* and *Account* in Feature Stub *Transaction*

the newly added method prototype in the feature stub. Consider again Figure 1 method `update` in feature *DailyLimit* of Figure 1. FEFALUTION creates a method prototype to match the call and replaces keyword `original` with the method prototype's name.

An addition to the original feature stub generation [31] is the introduction of abstract contracts for method prototypes. Otherwise, a theorem prover would assume the absence of a specification. As no definition for a method prototype exists (i.e., for method inlining), a (partial) proof cannot be established in this case. Therefore, FEFALUTION enriches method prototypes with *pure abstract contracts* (i.e., abstract contracts without concrete definitions), which we refer to as *contract prototypes*. Figure 6 shows the contract prototypes in Lines 16–18. Using contract prototypes, a theorem prover can now incorporate `update` in its analysis. Whenever a domain method calls a method prototype, the respective obligations can not be closed by the theorem prover and remain open. Consequently, a partial (incomplete) proof is generated that, however, already may contain numerous reusable proof steps. These *abstract sections* in partial proofs are replaced in the next phase by concrete definitions to generate full proofs.

3.3 Generation and Theorem Proving of the Metaproduct

Similar to feature stub generation, the metaproduct generation translates both the implementation and the specification. Following the variability encoding as described by Thüm et al. [38], FEFALUTION adds a new class `FeatureModel`,

in which each feature of the product line is represented by a static boolean field. During verification, the theorem prover can use these feature variables to simulate all valid feature combinations. To prevent the theorem prover from simulating invalid feature combinations, FEFALUTION adds an invariant to each class representing the feature model as a propositional formula.

Afterwards, FEFALUTION adapts the *domain methods* of all features in two ways. First, as briefly described in Section 3.1, *dispatcher methods* are introduced to increase robustness of the domain methods. In particular, FEFALUTION uses dispatcher methods to connect domain methods of all features along the feature composition order by creating a call hierarchy. Second, as FEFALUTION has access to the information of all features and the order of composition at this stage, keyword `original` is replaced by the concrete method call. In particular, a reference to `original` is only a special case of a feature-module dependency caused by a method call across different features. As an example, consider again method `update` in feature *DailyLimit* of Figure 1, which also contains keyword `original`. During the metaproduct generation, FEFALUTION replaces `original` with a call to the introductory method `update_BankAccount`, as can be seen in Line 22 of Figure 7.

To encode the whole product line's variability, FEFALUTION additionally adapts the specification. Again, we partly adopt the work of Thüm et al. [38]. They enrich the method contracts in their metaproduct with an implication before each clause defining under which feature combination it must hold. As stated above, their metaproduct methods are structurally similar to our *dispatcher methods*, so we adopt the same mechanism for our *dispatcher methods*. For *domain methods*, we mostly use the contracts from the feature-based phase as-is, including the additional **`requires`** clause stating that the corresponding feature must be selected. The only exception is keyword `original`. FEFALUTION is able to include all information from the product line. Therefore, FEFALUTION can replace keyword `original` with the precondition or postcondition of the respective method in the call hierarchy. This way, FEFALUTION is able to resolve all syntactic and semantic feature interactions. Finally, for the metaproduct, FEFALUTION transforms all method contracts into abstract contracts – which does not change the semantics of the contracts – to ensure reusability of the partial proofs.

In the metaproduct class in Figure 7, we also show some of the resulting contracts. For the *domain method* `update_DailyLimit`, the contract mostly stays as it was in the feature module. Only keyword `original` in the precondition and postcondition is replaced by the precondition and postcondition of the former refinement `update_BankAccount` (see Line 13 and Line 15). The contract of `dispatcher_update_DailyLimit` represents a composition of the contracts of *domain methods* `update_BankAccount` and `update_DailyLimit`. In front of each precondition and postcondition introduced by feature *DailyLimit*, there is an implication stating that the clause needs only to hold if the feature is selected (see Line 16). We do not need such an implication for feature *BankAccount*, because

```
1  public  class Account {
2      static final int OVERDRAFT_LIMIT = 0;
3      int balance = 0;
4      static final int DAILY_LIMIT = -1000;
5      int withdraw = 0;
6
7      /*@ ... @*/
8      boolean update_BankAccount(int x) {
9          [...]
10     }
11
12   /*@ requires_abs update_DailyLimitR;
13     @ def update_DailyLimitR = FM.FeatureModel.DailyLimit && x != 0;
14     @ ensures_abs update_DailyLimitE;
15     @ def update_DailyLimitE =\result <==> (balance == \old(balance) + x) &&
16     @  ((FM.FeatureModel.DailyLimit ==> (!\result ==> withdraw == \old(withdraw))
17     @  && (\result <==> withdraw <= \old(withdraw))));
18     @ assignable_abs update_DailyLimitA;
19     @ def update_DailyLimitA = withdraw, balance; @*/
20     boolean update_DailyLimit(int x) {
21         [...]
22         if (!update_BankAccount(x))
23             return false;
24         [...]
25     }
26
27   /*@ requires_abs dispatch_update_DailyLimitR;
28     @ def dispatch_update_DailyLimitR = (FM.FeatureModel.BankAccount ||
29     @  FM.FeatureModel.DailyLimit) && x != 0;
30     @ ensures_abs dispatch_update_DailyLimitE;
31     @ def dispatch_update_DailyLimitE = \result
32     @  <==> (balance == \old(balance) + x);
33     @ assignable_abs dispatch_update_DailyLimitA;
34     @ def updateA = withdraw, balance; @*/
35     boolean dispatch_update_DailyLimit (int x) {
36         if (FM.FeatureModel.DailyLimit)
37             return update_DailyLimit(x);
38         return update_BankAccount(x);
39     }
40
41     /*@ ... @*/
42     boolean update(int x) {
43         if (FM.FeatureModel.Logging)
44             return update_Logging(x);
45         return dispatch_update_DailyLimit(x);
46     }
47
48     /*@ ... @*/
49     boolean update_Logging(int x) {
50         [...]
51     }
52 }
```

Fig. 7. Class *Account* after Metaproduct Generation

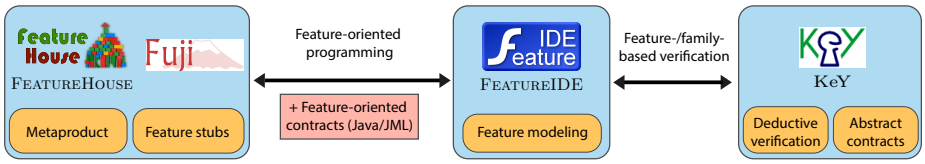

Fig. 8. Integration of FEFALUTION into the FEATUREIDE Ecosystem

this feature is part of any program variant. Finally, we add a precondition stating that at least *BankAccount* and *DailyLimit* must be selected (see Lines 28–29).

After the metaproduct generation, the partial poofs can be replayed on the adapted domain methods. If a proof goal remains open, this may be due to several reasons. First, a method may not fulfill its specification. In this case, either the behavior of the method or its specification needs to be changed. Second, if the contract correctly describes a method's behavior, but the theorem prover can still not close all proof goals, it might not be able to perform the necessary steps to complete the verification automatically. Still, interacting with the theorem prover may be possible. If all proof goals for the metaproduct are closed during verification, the product line is successfully verified.

4 Open-Source Tool Support

We implemented FEFALUTION as extensions to the tools FEATUREIDE and FEATUREHOUSE. FEATUREHOUSE [8] is a composer of software artifacts that supports feature-oriented composition for several languages. It is integrated into FEATUREIDE, an Eclipse-based IDE for feature-oriented product lines. Both tools have been extended to support (1) JML contracts and (2) variability encoding by means of the metaproduct generation technique proposed by Thüm et al. [38]. As we adopt some mechanisms of Thüm et al. [38] and rely on JML-based specification, we provide our tool support only as *extensions* to these tools. Moreover, one goal was to generate our verification objects (i.e., feature stubs and metaproduct) in such a way that they can be verified by any off-the-shelf theorem prover supporting JML contracts. However, for our approach we rely on abstract contracts and, to the best of our knowledge, only KEY provides them. Our last extension therefore integrates KEY into FEATUREIDE. In Figure 8, we illustrate how all three tools are connected.

When a product line is to be verified, the feature-stub generation can be started for any FEATUREHOUSE project in FEATUREIDE. The family-based type check is performed automatically before the actual generation is performed by means of the tool Fuji. Fuji [6] is a compiler for feature-oriented programming but also supports family-based type checking based on a family-wide access model. After the feature-stub creation, if KEY is installed as a plugin, it is started automatically with the first feature stub loaded. A user can employ KEY's strategy macro *Finish abstract proof part* to reason about abstract contracts, which results in

partial proofs based on the placeholders declared in the contract prototypes. Besides performing the actual verification, KeY can also save the created partial proofs in proof files on hard disk. When KeY is closed, FeatureIDE starts a new KeY instance with the next feature stub to bypass loading each feature stub manually. To start the second phase, one needs to re-build the product line to yield the current metaproduct. After the generation, the metaproduct can be verified with KeY. For *domain methods*, the partial proofs can be reused by employing KeY's proof replay feature and closing all remaining proof goals with KeY. For *dispatcher methods*, no partial proofs are generated. After verifying all methods of the metaproduct, the product line is considered to be completely verified. Both the base tools and our extensions are open-source and available at their respective repositories.[1] [2]

5 Empirical Evaluation of Fefalution

With Fefalution and our given tool support, we introduced a feature-family-based verification approach, which is intended to outperform existing product-line verification approaches considering the evolution of software product lines. The above sections raise the following two important research questions that we aim to answer by means of an empirical study.

RQ–1: Does Fefalution reduce the overall verification effort considering product-line evolution compared to existing approaches?
RQ–2: Which impact do different optimizations of family-based verification approaches have on the verification effort?

Answering **RQ–1** is important to understand whether our instance of a feature-family-based approach (i.e., Fefalution) is indeed a promising alternative to sole family-based approaches. Answering **RQ–2** will help users and researchers to get insights on concrete optimizations (e.g., employing either concrete or abstract contracts, or applying proof replay) that influence the verification effort.

5.1 Case Study

Our experiment is based on the bank account product line (cf. Figure 3) that has already been used for product-line specification and verification [42]. As Fefalution focuses on evolution, we developed a total of six different versions of the product line that each represent a common type of evolution scenario. All methods in each scenario are specified and can be verified automatically. We show all versions and how they are created in Figure 9.

Scenario S_1 represents our *base line* for each evolution scenario. Scenarios S_2 and S_3 represent evolution on the implementation level by either making changes (i.e., refactorings) to the contracts or the implementation. Scenarios S_4 and S_5

[1] Adapted FeatureHouse: https://github.com/kruegers/featurehouse
[2] Adapted FeatureIDE: https://github.com/kruegers/featureide

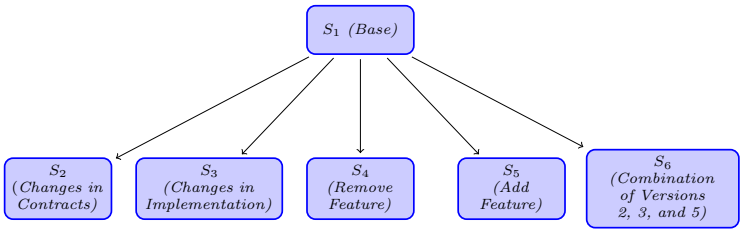

Fig. 9. Illustration of the Five Performed Evolution Scenarios

represent the evolution of the feature model. In particular, we remove feature *CreditWorthiness* for scenario S_4 and add feature *Logging* to the product line for scenario S_5. Finally, scenario S_6 combines the changes of scenarios S_2, S_3, and S_5.

The product line for scenario S_5 has already been presented in Figure 3. Overall, the product line for scenario S_5 consists of five classes distributed over nine features, twelve class refinements, a total of 17 unique methods specified with a contract, and six method refinements.

5.2 Experimental Design

We evaluate a total of six approaches by means of our existing tool support (cf. Section 4). As product-based approaches are typically inferior to family-based approaches [40], we only consider approaches that follow a family-based strategy. In particular, there exist numerous optimizations between the family-based verification approach developed in earlier work by some of the authors [38] and the feature-family-based verification approach developed in this work (i.e., FEFALUTION). Hence, to get more insights on the influence of these optimizations, we contribute four additional strategies to our comparison, where we alter some of the optimizations. We illustrate the six approaches in Table 1. Optimizations include whether (1) a feature-based phase exists, (2) abstract contracts are used to allow the creation of partial proofs, (3) method calls are treated either with inlining or contracting, (4) dispatcher methods are generated for the metaproduct to increase robustness to implementation changes, (5) proof replay is applied on features, and (6) proof replay is applied on the metaproduct.

We compare all six approaches according to our two research questions. To answer **RQ–1**, we examine the overall verification effort of all approaches. Verification effort is measured in terms of necessary proof steps, which we consider to be an adequate measurement in terms of proof complexity, and proof time in milliseconds. To answer **RQ–2**, we present the results from the perspective of product line evolution and discuss which optimizations of a verification approach have the most impact. Afterwards, we discuss the potential reuse of verification results. For the evaluation, we used a notebook with Intel Core i7-3610QM CPU @ 2.30GHz with 12 GB RAM on Windows 10 and Java 1.8.

Table 1. Evaluated Approaches and their Optimizations

Approach	Feature-Based Phase	Abstract Contract	Method Call Treatment with Contracting	Method Call Treatment with Inlining	Meta-Product Generation with Dispatcher	Meta-Product Generation without Dispatcher	Feature-Based Proof Replay	Family-Based Proof Replay
FEFALUTION	●	●	●	○	●	○	●	○
VA_2 (Metaproduct)	○	●	●	○	●	○	○	●
VA_3 (Concrete)	○	○	●	○	●	○	○	●
VA_4 (Inlining 1)	○	○	○	●	●	○	○	●
VA_5 (Inlining 2)	○	○	○	●	○	●	○	●
VA_6 (Family-based [38])	○	○	○	●	○	●	○	○

●: applied; ○ not applied

5.3 Results

We present the results in the following tables. In Table 2, we show the overall verification effort for each approach to verify all six versions (i.e., necessary proof steps, branches, and proof times) without depicting the reuse potential. In the following, we mainly discuss proof steps, as proof times and number of branches largely mirror the results and lead to similar interpretations.

Table 2. Overall Verification Effort for All Approaches

Approach	Proof Steps	Proof Time (in ms)	Branches
FEFALUTION	714,762	3,372,449	7,522
VA_2	665,994	2,762,019	5,041
VA_3	363,713	598,756	6,376
VA_4	157,072	258,145	2,919
VA_5	140,492	173,387	2,995
VA_6	153,931	187,156	3,499

As Table 2 shows, FEFALUTION needs the most steps of all approaches for a full verification of all versions. Overall, FEFALUTION needs approximately 7% more steps than VA_2, 49% more than approach VA_3, and more than four times as much for the remaining versions VA_4, VA_5, and VA_6). Although the overhead itself is not a surprise to us, we did not expect the gap between using abstract

contracts and concrete contracts to become this large. While approach VA_2, which only consists of the family-based phase, also leads to an overhead compared to the other approaches, it nevertheless needs less effort compared to FEFALUTION.

Table 3. Overall Verification Effort Considering Proof Reuse

Approach	Saved Proof Steps	Percentage of Reused Proof Steps	Percentage of Reused Branches
FEFALUTION	128,258	17.94%	44.65%
VA_2	129,396	19.43%	54.48%
VA_3	22,920	6.30%	10.24%
VA_4	13,212	8.41%	14.38%
VA_5	13,707	9.75%	16.69%

Regarding the overall proof reuse potential, Table 3 shows that about 128,258 steps (17.94%) of the total proof steps needed by FEFALUTION could be reused. Compared to FEFALUTION and VA_2, the reuse potential for all other approaches is considerably smaller. To get more insights about which approach performs better on which evolution scenario, we decided to conduct a more fine-grained analysis for the reuse potential. As described before, evolution scenarios S_2 and S_3 represent additions and changes to the *implementation and specification*, whereas S_4 and S_5 represent more coarse-grained changes to the *product line* (e.g., removing a complete feature module or other changes to the feature model). In the following, we investigate both kinds of evolution individually for all six approaches.

Table 4. Reuse for Versions with Changes in Implementation and Specification

Approach	Saved Proof Steps	Percentage of Reused Proof Steps	Percentage of Reused Branches
FEFALUTION	29,161	21.70%	60.97%
VA_2	29,506	23.91%	65.12%
VA_3	19,941	44.43%	33.01%
VA_4	10,630	50.47%	69.16%
VA_5	10,716	57.50%	79.06%

Table 4 shows that for changes that do not affect the feature model but only the implementation and specification, VA_4 and VA_5 are the most successful approaches with a reuse potential of over 50% each. The reuse potential for FEFALUTION and VA_2 is considerably smaller. However, when the feature model changes, as indicated by Table 5, the reuse potential for approaches VA_3, VA_4, and VA_5 drop significantly, whereas FEFALUTION and VA_2 perform significantly better

compared to all other approaches. Moreover, the reuse potential for FEFALUTION and VA_2 is similar in magnitude.

Table 5. Reuse for Versions with Changes to the Feature Model

Approach	Saved Proof Steps	Percentage of Reused Proof Steps	Percentage of Reused Branches
FEFALUTION	49,439	21.47%	63.06%
VA_2	50,568	22.63%	67.82%
VA_3	1,959	1.45%	3.11%
VA_4	1,703	2.58%	5.32%
VA_5	1,870	3.18%	6.87%

5.4 Discussion

For RQ1, we conclude that FEFALUTION reveals a large overhead compared to most approaches when considering the total verification effort. When the feature model evolves, FEFALUTION achieves a higher proof reuse than approaches VA_3, VA_4, and VA_5. However, VA_2 trumps FEFALUTION both in overall verification effort and proof reuse.

The proof strategies for abstract contracts are the same as for standard reasoning. This leads to some inefficient behavior when constructing partial proofs. For instance, once the program has been symbolically executed on a branch, there are several formulas of the form `if (locset \in method_A(...)) \then phi1 \else phi2`. As `method_A` is the abstract placeholder for the assignable clause, the conditional formula cannot be simplified further and leads to a proof split. Hence, we get 2^n branches for n such formulas.

When constructing the full proofs based on these partial proofs, this means we have to show the same proof obligations for several of these branches. This is avoided in case of concrete contracts as in most cases when inserting the concrete assignable clause, the condition of the conditional formulas simplifies to true or false and hence no unnecessary proof splits occur. By improving the strategies for the partial proofs, by stopping once a program has been symbolically executed and the remaining proof goal is first-order only, the proof size reduces drastically. A simple manual emulation of such a proof strategy leads to significant improvements. For instance, for method `transfer` of feature *Transaction* in S_6, the proof size is reduced from 111,075 nodes to 34,259 nodes. Further improvements by a more intelligent expansion of the placeholders may thus lead to further improvements.

Additionally, most features that had to be re-verified in the feature-based phase in S_2 to S_6 resulted in partial proofs containing less than 50 proof steps. The potential for feature-based proof replay was therefore limited in our case study.

RQ-1: Evaluation of Fefalution

For the bank account product line, FEFALUTION *revealed a slight overhead for each evolution scenario compared to the sole family-based approach (i.e., VA_2). However, our manual analysis shows also room for improvement in two directions. First, the internal expansion of abstract contracts can be improved to drastically reduce the overhead. Second, feature-based proof replay was only applicable to a limited degree, as many established partial proofs during our evaluation consisted of less than 50 proof steps. More complex evolution scenarios may lead to more significant reductions of verification effort.*

To answer RQ2, we compare FEFALUTION with five other approaches that each alter a specific *optimization* (cf. Table 1). For proof composition, our results indicate that a feature-family-based approach with partial proofs based on abstract contracts does not increase the reuse potential but, in fact, reduces it slightly when compared to a sole family-based approach with the same optimizations otherwise. To determine the impact of abstract contracts in contrast to concrete contracts, we can compare the results of approaches VA_2 and VA_3. Here, the results indicate that abstract contracts represent a trade-off between an overhead for a single verification on the one hand and an increased proof reuse on the other hand. Consequently, FEFALUTION and VA_2 need more effort for a full verification, but manage to achieve a much higher reuse than VA_3. This is not surprising, as abstract contracts were designed to facilitate more proof reuse, even if that leads to a small overhead [11]. Finally, we compare the results of approaches VA_3 and VA_4 to evaluate method call handling during verification. As illustrated, treating method calls with inlining instead of contracting leads to much lower overall verification effort, which may imply that our designed case study is insufficient to showcase the benefits of contracting compared to method inlining under evolution.

RQ-2: Comparison of all Approaches

For the bank account product line, abstract contracts lead to significant overhead regarding the overall verification effort, whereas approaches using method inlining require the least amount of verification effort. Considering only the feature model evolution, potential proof reuse is 8–10 times higher with abstract contacts. Contrary, considering only the evolution of implementation and specification, potential proof reuse is only half as high.

Additionally to these results, we made an observation during the evaluation, we want to discuss in the following. The smaller a full proof of a method is the bigger partial proofs tend to become relatively. For methods that need less than 100 proof steps for full verification, partial proofs often provide already more than 70% of the needed proof steps. For methods whose proofs are larger ($>$ 1000 proof steps), however, the partial proofs are often less than 10% of the steps needed for a full proof. Although the general notion is not surprising, the huge

drop is unfortunate because it reduces the usefulness of abstract contracts and partial proofs for software systems with methods for inherently more complex correctness proofs.

5.5 Further Insights and Future Directions

Our discussion in Section 5.4 implies that part of the results are indeed artifacts of the used *proof strategy*. Current program verification systems including KeY are mostly optimized for finding proofs. The feature-based phase together with the generated partial proofs, however, add an additional complexity to the proof search, which amounts to the overhead we measured for the bank account product line. This could very well be a design issue in current verification systems, which need to be adapted for a feature-based framework.

There is also another possibly fruitful direction to continue this line of work that has not been explored before. In recent work, Steinhöfel and Hähnle [36] suggest *abstract execution*, which has the potential to replace or at least enhance our usage of abstract contracts. With abstract execution (1) any statement or expression can be abstract, such that more fine-grained reuse than at the method level is possible, (2) abstract contracts can specify additional properties, such as necessity for termination or return values, which makes them more flexible, and (3) abstract contracts may contain dynamic frames and can express abstract heap separation conditions. Together, this has the potential to shift the trade-off in favor of our feature-family-based approach, as abstract contracts become richer. For example, the abstract contract of the logging feature would contain a dynamic frame that forces the assignable heap logger code to be disjoint with other methods. This means one can prove at the feature level that logging does not interfere with other features. Upon composition, the dynamic frame must be instantiated and proven, but this is usually trivial.

6 Related Work

In a survey, Thüm et al. [40] classify approaches in the literature for analysis of software product lines. They differentiate the approaches into product-based, family-based, feature-based approaches, and combinations thereof. Following their classification, we categorize our approach as a feature-family-based verification approach. Fefalution is first such approach without restrictions on specification. We provide the first comparative evaluation of a feature-family-based approach.

Product-based approaches usually require the generation of all products. Harhurin and Hartmann [27] among others propose optimizations such as verifying only a base product and reusing the proofs for the other products. However, even for optimized approaches, for product lines with many products, too many products need to be generated, which is why we chose not to include a product-based phase in our approach.

To facilitate feature-based approaches, combinations with product-based approaches have been proposed. Thüm et al. [37], Damiani et al. [18], Delaware

et al. [20, 21] and Gondal et al. [23] propose approaches that first create partial proofs for all features/implementation units and then compose these to full proofs for all products. Although our approach also consists of a feature-based phase to produce partial proofs and we compose the partial proofs in a second phase, our approach performs the composition in a family-based phase instead of a product-based phase.

Thüm et al. [38] present a family-based approach, which creates one metaproduct by means of variability encoding. FEFALUTION is similar as its second phase is also family-based, and we adopt some parts of the metaproduct generation. FEFALUTION differs in that it uses a different form of variability encoding and consists of two phases to facilitate proof reuse and to incorporate the notion of evolution. We also evaluate FEFALUTION in comparison to other family-based approaches instead of comparing it to product-based approaches.

In [17] the authors propose a counter example-guided approach (CEGAR) to static analysis and verification of metaproducts as presented in [38]. Their approach applies two kinds of refinements: the first decomposes the analysed metaproduct in two parts, one part for which a found counterexample is not present and one where it is. Both parts can then be analyzed independently, but each of which has a reduced overall complexity. The second one "refines" the analysis tool. The idea is to start with a more efficient, but less powerful static analyses and to switch to e.g. a full verification systems for those parts for which the less powerful analyses fails. In contrast to FEFALUTION their approach is not based on reuse, but achieves scalability by decomposition and applying more powerful, but less automatic tools on less complex parts.

Hähnle and Schaefer [25] propose to apply the Liskov Principle to contracts in order to achieve a feature-family-based verification. Their approach requires that when a method's pre- and postconditions are modified in a delta, they must become more specific than in the original implementation unit. Similarly, assignable clauses in deltas are only allowed to be subsets of the original assignable clause. Consequently, their approach allows for a modular verification and thus a reuse of old still valid proofs but the restrictions it imposes on contract refinement limit its practicability. With FEFALUTION, we aim to achieve a similarly modular verification without restrictions on contracts.

On top of the discussed differences, none of the approaches consider product-line evolution and therefore often require a complete re-verification. In order to facilitate proof reuse, Hähnle et al. [26] propose abstract contracts. In their approach, abstract contracts are used to provide placeholders independent of the actual definition of the contracts. A theorem prover can create a partial proof with respect to the placeholders that can be used for a full verification when the concrete definitions are known. Bubel et al. [11] further explore and extend the concept and provide tool support. Bubel et al. [12] also propose the concept of *proof repositories* to explicitly address the problem of software evolution and inefficient re-verification by employing abstract contracts. Contrary to their results, applying abstract contracts to the evolution of features seems to produce a higher overhead in its current stage.

Feature-family-based approaches have also been applied to other verification techniques than theorem proving. Delaware et al. [19] define a type system based on a feature-aware subset of Java. Type safety of individual features is specified through constraints, which can then be used to guarantee the whole family's type safety by relating them to the feature model. Damiani and Schaefer [16] propose a feature-family-based type checking approach for delta-oriented product lines. They manage to partially type-check each delta in isolation before using these results for a full family-based type check. However, type checking is not sufficient to detect semantic feature interactions. Hence, we designed our approach as a deductive verification approach.

7 Conclusions

Software product-line engineering facilitates a paradigm for systematically developing a set of program variants with a common code base. To verify program variants, numerous product-based and family-based strategies were proposed over the last decade. The lack of addressing feature interactions make sole feature-based strategies less effective, but they also exhibit the potential to scale better than sole family-based approaches. The goal of this work was to systematically investigate how to reduce the verification effort of specified product lines under evolution, whilst the focus of our discussion lies on method contracts and feature-oriented programming.

To this end, we conducted an empirical study in which we compared six approaches with varying characteristics. Our evaluation was based on an existing benchmark, namely the bank account product line, comprising a base version and five common evolution scenarios. Besides measuring and discussing the overall verification effort for each scenario, we confirmed that abstract contracts are mostly valuable when the product lines evolves (e.g., adding new features). However, as method inlining performed better than contracting, we also conclude that our employed case study lacked complexity in order to draw many significant conclusions. Our proposed feature-family-based verification FEFALUTION also performed slightly worse than the sole family-based verification, which we attribute to the limited applicability of the feature-based proof replay.

Nonetheless, we argue that our initial study and tooling constitutes a first stepping stone for a more thorough investigation of the verification of evolving product lines. While the lack of adequate case studies and benchmarks is certainly averse to our initial goal, our automatic tooling can be employed as-is to continue this line of research. Questions such as *Can* FEFALUTION *outperform the family-based strategy for any evolution scenario?* or *Will a combination of feature-based proof replay and family-based proof replay reduce the overall verification effort?* are particularly interesting to investigate in future work.

Acknowledgements. We are grateful to Stefanie Bolle for her help with the implementation and evaluation, and also to Dominic Steinhöfel for his support with KeY and abstract contracts. This work was supported by the DFG (German Research Foundation) under the Researcher Unit FOR1800: Controlling Concurrent Change (CCC).

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification–the key book. *Lecture Notes in Computer Science*, 10001, 2016.

[2] Sven Apel and Delesley Hutchins. A Calculus for Uniform Feature Composition. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 32 (5):19:1–19:33, May 2010.

[3] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In *Proc. Int'l Symposium Software Composition (SC)*, pages 20–35, 2008.

[4] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231, Washington, DC, USA, 2009. IEEE. ISBN 978-1-4244-3453-4.

[5] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[6] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming (SCP)*, 77(3):174–187, 2012.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013.

[8] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79, JAN 2013.

[9] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491, Piscataway, 2013. IEEE. ISBN 978-1-4673-3076-3.

[10] Lerina Aversano, Massimiliano Di Penta, and Ira D. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. Int'l Working Conference Source Code Analysis and Manipulation (SCAM)*, pages 83–92, Washington, DC, USA, October 2002. IEEE. ISBN 0-7695-1793-5.

[11] Richard Bubel, Reiner Hähnle, and Maria Pelevina. Fully abstract operation contracts. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, LNCS. Springer, 2014.

[12] Richard Bubel, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu. Proof repositories for compositional verification of evolving software systems. In *Transactions on Foundations for Mastering Change I*, pages 130–156. Springer, 2016.

[13] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

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

[15] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA, 2006. ACM.

[16] Ferruccio Damiani and Ina Schaefer. Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 193–207, Berlin, Heidelberg, October 2012. Springer. ISBN 978-3-642-34025-3.

[17] Ferruccio Damiani, Reiner Hähnle, and Michael Lienhardt. Abstraction refinement for the analysis of software product lines. In Sebastian Gabmeyer and Einar Broch Johnsen, editors, *Tests and Proofs - 11th International Conference, TAP 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, volume 10375 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017. https://doi.org/10.1007/978-3-319-61467-0_1. URL https://doi.org/10.1007/978-3-319-61467-0_1.

[18] Feruccio Damiani, Johan Dovland, Einar Broch Johnsen, Olaf Owe, Ina Schäfer, and Ingrid Chieh Yu. A transformational proof system for delta-oriented programming: Proceedings of the 16th international software product line conference. In Santana de Almeida, Eduardo, editor, *Proc. Int'l Software Product Line Conf. (SPLC)*, volume 2, pages 53–60, New York and NY and USA, 2012. ACM. ISBN 978-1-4503-1095-6.

[19] Benjamin Delaware, William R Cook, and Don Batory. Fitting the pieces together: a machine-checked model of safe composition. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 243–252. ACM, 2009.

[20] Benjamin Delaware, William Cook, and Don Batory. Product lines of theorems. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 595–608, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0.

[21] Benjamin Delaware, d. S. Oliveira, Bruno C., and Tom Schrijvers. Meta-theory à la carte. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 207–218, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7.

[22] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proc. Int'l Workshop Role of Software Architecture for Testing and Analysis (ROSATEA)*, pages 39–48, New York, NY, USA, 2006. ACM.

[23] Ali Gondal, Michael Poppleton, and Michael Butler. Composing event-b specifications: Case-study experience. In *Proc. Int'l Symposium Software*

Composition (SC), pages 100–115, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-22044-9.

[24] Alexander Gruler, Martin Leucker, and Kathrin 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, 2008. Springer. ISBN 978-3-540-68862-4.

[25] Reiner Hähnle and Ina Schaefer. A Liskov principle for delta-oriented programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 1, pages 32–46, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[26] Reiner Hähnle, Ina Schaefer, and Richard Bubel. Reuse in software verification by abstract method calls. In *Proc. Int'l Conf. Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 300–314, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-38573-5.

[27] Alexander Harhurin and Judith Hartmann. Towards consistent specifications of product families. In *Proc. Int'l Symposium Formal Methods (FM)*, pages 390–405, Berlin, Heidelberg, 2008. Springer. https://doi.org/10.1007/978-3-540-68237-0_27.

[28] Peter Höfner, Bernhard Möller, and Andreas Zelend. Foundations of coloring algebra with consequences for feature-oriented programming. In *Proc. Int'l Conf. Relational and Algebraic Methods in Computer Science (RAMiCS)*, pages 33–49. Springer, 2012.

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

[30] Alexander Knüppel, Thomas Thüm, Carsten Padylla, and Ina Schaefer. Scalability of deductive verification depends on method call treatment. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 159–175. Springer, 2018.

[31] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 115–124, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2373-4.

[32] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, September 2006. URL http://www.jmlspecs.org/jmldbc.pdf.

[33] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[34] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10): 40–51, 1992.

[35] David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, December 1972.

[36] Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In *Proc. Int'l Symposium Formal Methods (FM)*, pages 319–336. Springer, 2019.

[37] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, 2011. IEEE Computer.

[38] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA, September 2012. ACM.

[39] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying design by contract to feature-oriented programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212, pages 255–269, Berlin, Heidelberg, 2012. Springer.

[40] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[41] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, January 2014.

[42] Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. Feature-oriented contract composition. *Journal of Systems and Software*, 152:83–107, 2019.

[43] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1):125–145, January 2016.