Proof Composition for Deductive Verification of Software Product Lines

Thomas Thüm*, Ina Schaefer†, Martin Kuhlemann*, and Sven Apel‡
*University of Magdeburg, Germany
†University of Braunschweig, Germany
‡University of Passau, Germany

***Abstract*—Software product line engineering aims at the efficient development of program variants that share a common set of features and that differ in other features. Product lines can be efficiently developed using feature-oriented programming. Given a feature selection and the code artifacts for each feature, program variants can be generated automatically. The quality of the program variants can be rigorously ensured by formal verification. However, verification of all program variants can be expensive and include redundant verification tasks. We introduce a classification of existing software product line verification approaches and propose proof composition as a novel approach. Proof composition generates correctness proofs of each program variant based on partial proofs of each feature. We present a case study to evaluate proof composition and demonstrate that it reduces the effort for verification.**

***Keywords*-Software product lines, proof composition, feature-based verification, JML.**

I. Introduction

A software product line (SPL) is a set of software-intensive systems that share code [1]. The program variants of an SPL are distinguished in terms of features [2]. The idea of software product line engineering is to generate similar programs from a common code base simply by specifying the desired features. The quality of the generated program variants can be rigorously ensured by formal verification. However, the number of programs that can be generated from an SPL is up-to exponential in the number of features. Hence, formal verification of every program using state-of-the-art verification tools is not feasible [3], [4]. Instead, for the verification of SPLs, verification effort should be reused in the same way as with code.

We give an overview of how to reduce the effort in verifying all programs of an SPL. We classify existing approaches and argue that there is a need for more sophisticated techniques. Based on these insights, we propose proof composition as a technique to achieve high reuse and thus efficiency in the verification of SPLs.

We concentrate on deductive verification in which correctness proofs are written by humans. Previous work in this direction used model checking. Both approaches have benefits and are complementary to each other. Model checking can be applied fully automatically. But with user interaction and deductive verification we may be able to prove much stronger properties about SPLs, which cannot be proven using model checking.

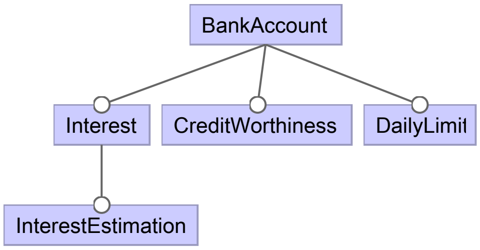

Figure 1. A feature model of an SPL of bank accounts

The idea of proof composition is to create a proof base. Similarly to a code base, proofs are modularized into parts that are specific to certain features. Given a particular feature selection, we do not only generate the program variant, but additionally, we compose a proof for the program variant from the partial proofs associated with the features involved. A proof assistant can check whether the composed proof is valid and thus verify that the program variant fulfills the desired specification.

We use an SPL of bank accounts as an illustrative case study. The SPL is developed using feature-oriented programming where the program variants are written in Java. Program specifications are expressed using the Java Modeling Language (JML) [5]. We use Why and Krakatoa [6] to generate proof obligations for the proof assistant Coq [7]. We analyze and demonstrate how proofs written in Coq can be composed.

II. Background and Motivating Example

In this section, we introduce the concept of SPLs and present the program variants included in our bank account product line. We show how the example SPL is implemented using feature-oriented programming and define how program properties are specified using design by contract [8].

A. Software Product Lines

An SPL is a set of program variants. SPL engineering is in particular used for program variants of the same domain, e.g., the domain of bank account software. The advantage of SPL engineering is that common code in multiple program variants can be reused. We distinguish the program variants of an SPL by their features. A feature is an end-user–visible

```
class Account {
  //@ invariant balance_non_negative: balance>=0;
  int balance = 0;
  //@ ensures balance == 0;
  Account() {}
  /*@
   @ ensures (!(\old(balance)+x >= 0) || \result)
   @ && ((\old(balance)+x >= 0) || !\result)
   @ && (\result || balance == \old(balance))
   @ && (!\result || balance == \old(balance)+x);
   @*/
  boolean update(int x) {
    int newBalance = balance + x;
    if (newBalance < 0)
      return false;
    balance = newBalance;
    return true;
  }
}
class Application {
  Account account = new Account();
  void nextDay() {}
  void nextYear() {}
}
```

Figure 2. Feature module for feature *BankAccount*

program characteristic [2], e.g., whether a bank account supports a daily limit for withdrawing money.

Not all combinations of features are meaningful for a given domain. For example, consider an SPL for the management of bank accounts. A client may get interest or not and the software may be able to estimate the interest for the current year. Having the feature interest estimation without support for interest at all does not seem useful. Hence, we need to specify the valid combinations of features for an SPL, which is usually defined by a feature model. A feature model [2] documents all features and the constraints between them, e.g., every feature requires its parent feature.

In Figure 1, we show the feature model of our running example. The feature *BankAccount* is present in all program variants and represents our base implementation. All other features can optionally be selected, independent of each other, except that *InterestEstimation* requires the feature *Interest*.

Given the set of features F, a program variant P is determined by a subset of all features (i.e., $P \subseteq F$). An SPL S is a set of program variants P_i, i.e., $S = \{P_1, P_2, \dots, P_n\}$ and $S \subseteq 2^F$. In our example, the features *Interest*, *CreditWorthiness*, and *DailyLimit* can be selected independently of each other resulting in $2^3 = 8$ combinations. Half of these combinations contain the feature *Interest* and we can additionally choose whether the feature *InterestEstimation* is contained or not, resulting in $4{*}2{+}4 = 12$ program variants. Already for this small set of features, it is laborious to verify each program variant separately.

```
refines class Account {
  final static int DAYLY_LIMIT = -1000;
  //@ invariant withdraw_in_limit:
  //@ withdraw >= DAYLY_LIMIT;
  int withdraw = 0;
  /*@
   @ ensures (\old(withdraw)+x >= DAYLY_LIMIT)
   @ || !\result;
   @*/
  refines boolean update(int x) {
    if (x < 0)  {
      int newWithdraw = withdraw + x;
      if (newWithdraw < DAYLY_LIMIT)
        return false;
      withdraw = newWithdraw;
    }
    return original(x);
  }
}
refines class Application {
  /*@
   @ ensures account.withdraw == 0;
   @*/
  refines void nextDay() {
    original();
    account.withdraw = 0;
  }
}
```

Figure 3. Feature module for feature *DailyLimit*

B. Feature-Oriented Programming

SPLs can be implemented using feature-oriented programming (FOP) [9]. In FOP, we decompose code artifacts into pieces belonging to individual features, i.e., a class is split into several files containing some of the fields and methods. A feature module encapsulates all code artifacts concerning one feature.

In Figure 2, we present the feature module implementing feature *BankAccount*. It contains the classes `Account` and `Application`. Class `Account` basically consists of a field saving the balance of the account and a method to update the balance by a certain value. Method `update` ensures that the balance is always positive. The second class `Application` holds an object `Account` and provides two empty methods that handle the situation when a new day or new year is reached. These methods are empty, but refined subsequently by other feature modules.

A further feature module implementing the feature *DailyLimit* is shown in Figure 3. It refines the two classes introduced by feature *BankAccount*. Specifically, it adds a constant defining the daily limit, i.e., the highest possible withdrawal for one day. Furthermore, a new variable to save the current withdrawal of the day is added. The method `update` is refined to store the withdrawal and check whether it is within the daily limit.

A method defined in a certain feature module may refine a method from another module by overriding. The keyword `original` is used to refer to the extended method (see

Figure 3). The strength of FOP is that the feature modules can be used in all combinations allowed by the feature model.

C. Verification using Design by Contract

Critical requirements of programs can be specified using *design by contract* [8]. Each class is specified by a set of class invariants. A *class invariant* is a condition that must hold for all methods of the class. This means that if the invariant holds before executing a method, the invariant still has to hold after execution. Additionally, methods can be specified by *method contracts*, consisting of a precondition and a postcondition. A method contract is valid if assuming the precondition before method execution, the postcondition is established after the execution of the method.

We express specifications of Java program variants using the Java Modeling Language (JML) [5]. In JML, the keyword `requires` defines a precondition, whereas keyword `ensures` defines a postcondition. Empty pre- or postconditions are assumed to be always true and can be omitted. In pre- and postconditions, JML allows Java expressions, in which the keyword `old` references the value of a field before executing the method and `result` refers to the return value of the method. In Figures 2 and 3, examples for JML specifications are given. For instance, in Figure 2, Line 2, a class invariant is defined to ensure that the balance of an account is not negative. In Lines 3 and 4 of Figure 3, the class invariant requires that the withdraw is within the daily limit.[1]

Given a Java program with a specification in terms of JML annotations, it has to be proven that the program satisfies its specification. From a JML-annotated Java program, corresponding proof obligations in a proof assistant's language can be generated. A proof obligation is a theorem consisting of certain hypotheses and a conclusion referring to the JML specifications, e.g., that a constructor fulfills all invariants. In order to establish that the program satisfies the specification, proofs have to be written for every proof obligation and a proof assistant can check that the proofs are correct.

III. Verification of Software Product Lines

An SPL is a set of programs that share similarities, which raises the question of how to verify these programs efficiently. In principle, we could generate all program variants (24 in our example) and verify their correctness independently. But, this leads to highly redundant work as the programs share code and we have to prove several theorems multiple times. As proving correctness can usually not be done fully automatically for large programs, an efficient method to verify all programs of an SPL reduces not only calculation time, but also the man power needed to interactively prove correctness.

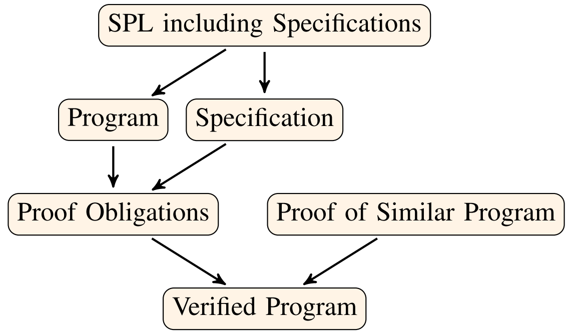

Figure 4. Variant-based verification using proof reuse

We introduce a classification of approaches for SPL verification. We distinguish variant-based and feature-based verification. The classification is closely related to the scalability of the approaches with respect to the number of program variants. Roughly speaking, the effort needed for variant-based verification is proportional to the number of variants, whereas feature-based verification growths in the number of features. Proof composition proposed in this paper can be classified as feature-based verification.

A. Variant-based Verification

The overall idea of variant-based verification is to generate all program variants and to verify each variant separately. In Figure 4, we illustrate this strategy. We start with an SPL and specifications for all features. Using the software generator, we can generate a certain program variant and its specification. Both are the input for a proof obligation generator. We retrieve a number of theorems which are then to be proven automatically or interactively. Current approaches for feature interaction detection [10] are instances of variant-based verification techniques. Single features are expressed in a formal specification language, e.g., see [11], and all pairs of potentially interacting features are checked.

With an increasing number of program variants, variant-based verification gets too complex. The reason is that we have to prove certain theorems over and over again, e.g., if more than one program variant contains the same method with the same specification. Hence, Bruns et al. presented an approach to reuse existing proofs across other program variants [12]. Their approach is based on delta-oriented programming. They write the proof for a particular program variant and reuse parts of it for another program variant by comparing the delta between the source code of both variants.

Variant-based verification is a generally applicable technique to verify SPLs. It can also be applied to stand-alone systems. But, there are several problems with this approach. How can we reuse the proofs across two or more program

[1] We do not need to force that the daily limit is always smaller than zero, because Why uses the value of constants directly at the proofs, i.e., changing -1000 to a positive integer would invalidate the proofs.

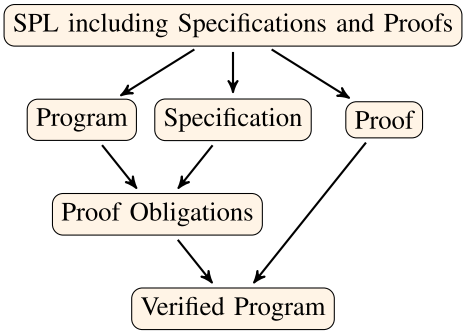

Figure 5. Feature-based verification using proof composition

variants? What is the best order in which to verify all program variants? We might find good solutions to these problems, but a more fundamental problem is that we need to redo all proofs if the source code or specification of a certain feature evolves.

B. *Feature-based Verification*

Feature-based verification aims at efficiently proving the correctness of all program variants of an SPL. The idea is to prove properties of features instead of variants to avoid redundant proof writing. Basically, there are two possibilities how to infer properties about features.

First, we could create and prove theorems for each feature and use them to prove the correctness of a program variant. This approach is pursued in [3], [4], where for each property that a feature should satisfy, constraints on the behavior of the other features are generated. If the composed features satisfy the constraints, the properties of the considered feature are maintained.

Second, we can write partial proofs as part of each feature that can be composed to more complex proofs for program variants. We pursue this novel approach in proof composition, since it alleviates the need to create additional specifications for features that have to be established when the feature modules are composed.

In Figure 5, we illustrate the feature-based verification process using proof composition. We generate program variants and specifications based on a feature selection and use them as an input for the generation of proof obligations. In addition, we generate correctness proofs for the given feature selection from the partial proofs in the proof base. Then, we merge the proof obligations with the generated proofs, and a proof assistant can check the validity of the composed proof. In Section IV, we show how proofs for the proof assistant Coq can be composed from partial proofs.

Feature-based verification appears to be more efficient than variant-based verification. The reason is that whatever we can prove for a feature in isolation, has to be proven only once. Variant-based verification may come to a similar result, but with more overhead to check whether the same theorem was already proved in any other variant.

IV. Proof Composition

We illustrate proof composition for feature-based verification of SPLs using a case study of bank accounts. The aim of proof composition is to write partial proofs for each feature and to compose the partial proofs of the selected features to retrieve the correctness proof of a certain program variant.

Proof composition relies on tools to generate proof obligations from a program and a specification. We use the tool chain Krakatoa/Why [6] and Coq [7]. The tool Why is a verification platform providing a language for proof obligations. Krakatoa is a plug-in for Why that parses a JML-annotated Java file and produces proof obligations in the Why language. Using Why, proof obligations can be exported to several proof assistants. We selected the proof assistant Coq because of our expertise in Coq.

A proof obligation in Coq consists of several premises and one conclusion. A manually written proof script transforms the conclusion into parts that are equivalent to the premises using proof steps. Coq provides a language for these proof steps and can verify their correctness. We cannot give here a full introduction to the proof assistant Coq and refer the reader to the Coq manual [7]. To follow our arguments, it is enough to understand the overall structure of a Coq proof obligation and associated proof.

In Figure 6, we present a proof obligation in Coq retrieved for the program variant {*BankAccount*} and a proof written by ourselves. The proof obligation states that the constructor of class `Account` ensures all class invariants, that is, all invariants hold after object construction. Figure 2 contains the corresponding JML-annotated Java code. Line 1 in Figure 6 contains a generated name for the theorem that is used to relocate the proof, if we change the source code and regenerate the proof obligations. Lines 2 to 9 describe premises that can be used to prove the conclusion in Line 10. The human-written proof steps are surrounded by `Proof` and `Save`.

In Figure 7, we show the same proof obligation for the program variant {*BankAccount*, *DailyLimit*}. It differs threefold from the one for the program variant {*BankAccount*}. First, we get new premises at the Lines 4 and 11–14. Second, we have a new conclusion to prove in Line 16. Third, the Lines 23–28 contain new proof steps proving the additional conclusion.

For proof composition, we are interested whether proofs can be modularized into features. Can we assign every proof step to one feature and generate it only if the feature is present in a program variant? The proof in Figure 7 can be modularized as follows. The Lines 24–29 belong to feature *DailyLimit*, whereas the Lines 19–23 concern feature *BankAccount*. We refer to these parts as partial proofs.

Given the partial proofs for each feature, we now define how to compose these partial proofs and how this fits into the tool infrastructure using Krakatoa and Why.

```
1  (*Why goal*) Lemma cons_Account_safety_po_1 :
2  forall (this_1: (pointer Object)),
3  forall (Account_balance: (memory Object int32)),
4  forall (Object_alloc_table:(alloc_table Object)),
5  forall (HW_1: (valid_struct_Account this_1 0 0
       Object_alloc_table)),
6  forall (result: int32),
7  forall (HW_2: (integer_of_int32 result) = 0),
8  forall (Account_balance0: (memory Object int32)),
9  forall (HW_3: Account_balance0 = (store
       Account_balance this_1 result)),
10 (* JC_17 *) (balance_non_negative this_1
       Account_balance0).
11 Proof.
12 intuition.
13 unfold balance_non_negative.
14 replace Account_balance0 with (store
       Account_balance this_1 result).
15 rewrite select_store_eq; trivial.
16 omega.
17 Save.
```

Figure 6. Constructor of class `Account` ensures invariants in program variant {*BankAccount*}

In Why, proof obligations are located by their names to support the evolution of software. Assume we have already written a proof for a particular program and need to apply changes to this program. In this scenario, Why is able to regenerate the proof obligations while the hand-written proof scripts remain untouched.

Hence, we can store partial proofs with their names, and Why is able to generate the proof obligations for a particular program variant. In Figure 6, the partial proof for feature *BankAccount* consists of Line 1 containing the name and the Lines 11–17 with the proof steps. In proof composition, partial proofs are composed based on their names. Partial proofs with identical names are composed by concatenation.

A. Decomposability of Proofs

So far, we only considered an example proof obligation when adding a particular feature. Now, we want to systematically consider all possible cases how a feature can transform the source code and its specification as well as the corresponding changes of the proof obligations.

A feature module can add a field with or without an invariant, a method with or without a method contract, or refine an existing method. For the refinement of an existing method, we only allow compatible contracts. A contract A is compatible to a contract B, if whenever a method m satisfies A, it also satisfies B. A method refinement usually maintains the behavior of the overridden method, so the compatibility between contracts is given. As a result, all proofs relying on a method's contract remain valid when refining the method with a compatible contract.

In the following, we discuss how proof obligations may change when a new feature is added. We abstract from technical issues, which we discuss later.

```
(*Why goal*) Lemma cons_Account_safety_po_1 :
forall (this_3: (pointer Object)),
forall (Account_balance: (memory Object int32)),
forall (Account_withdraw: (memory Object int32)),
forall (Object_alloc_table:(alloc_table Object)),
forall (HW_1: (valid_struct_Account this_3 0 0
    Object_alloc_table)),
forall (result: int32),
forall (HW_2: (integer_of_int32 result) = 0),
forall (Account_balance0: (memory Object int32)),
forall (HW_3: Account_balance0 = (store
    Account_balance this_3 result)),
forall (result0: int32),
forall (HW_4: (integer_of_int32 result0) = 0),
forall (Account_withdraw0:(memory Object int32)),
forall (HW_5: Account_withdraw0 = (store
    Account_withdraw this_3 result0)),
(* JC_45 *) ((balance_non_negative this_3
    Account_balance0)
∧ (withdraw_in_limit this_3 Account_withdraw0)).
Proof.
intuition.
unfold balance_non_negative.
replace Account_balance0 with (store
    Account_balance this_3 result).
rewrite select_store_eq; trivial.
omega.
unfold withdraw_in_limit.
replace Account_withdraw0 with (store
    Account_withdraw this_3 result0).
rewrite select_store_eq; trivial.
replace (integer_of_int32 result0) with 0.
unfold Account_DAILY_LIMIT.
rewrite int32_coerce; omega.
Save.
```

Figure 7. Constructor of class `Account` ensures invariants in program variant {*BankAccount*, *DailyLimit*}

- *Adding a new field without an invariant.* We get no new proof obligations for any class. A new premise that represents the initialization of the field is added to the proof obligations for constructors (see Case 2 below).
- *Adding a new field with an invariant.* We need to prove for every method and constructor of the class that the invariant is fulfilled after execution. This changes proof obligations concerning constructors and existing methods (Case 3). New proof obligations are generated in case there were no invariants before (Case 1).
- *Adding a new method with or without a contract.* No existing proof obligations are changed. But, if the class contains invariants or if the method has a contract, we get new proof obligations that the method fulfills the the existing invariants or the newly added contract (Case 1).
- *Refining a method with a compatible contract.* No existing proof obligations are changed, because we only allow to refine a method with a compatible contract. We get new proof obligations to show that the refined method fulfills the existing invariants and the added compatible contract (Case 1). A further new proof

obligation is needed to show that the contract is indeed compatible (Case 1).

We analyzed how proof obligations can change for particular changes induced by a feature. Now, we go through all possible changes to proof obligations and consider how the proofs need to be changed accordingly.

Case 1 *New proof obligations.* For every newly created proof obligation, we can write a new partial proof for the new feature. Proof composition simply copies it to the proof of the composed program variant.

Case 2 *New premises at proof obligations.* A new premise does not imply a change to the proof steps.

Case 3 *New cases at proof obligations.* The changed proof obligations contain a new conclusion for which we need additional proof steps at the end of the proof. This can be handled as shown in the above example, i.e., creating a new partial proof including these additional proof steps. Proof composition will concatenate the partial proofs for program variants.

B. Technical Issues

The previous case distinction shows that proof composition is generally possible. Next, we want to discuss some technical problems we faced and give suggestions for solutions.

First, whenever composing a module introducing a field with an invariant, it should be added below all other fields. This simplifies proof composition, since new cases always appear at the end of proof obligations and new proof steps can always be added below existing proof steps.

Second, Krakatoa and Why generate names for premises, assignments, nested expressions, and for the current object. These names are needed to reference these entities in proof steps. The problem is that name generation is very fragile in terms of changes to the source code. Hence, we suggest to minimize references to generated names, which is possible to a certain degree. Remaining references can mostly be updated automatically based on the generated proof obligation.

Third, in FOP, refined methods are usually renamed and the keyword `original` is replaced by a call to that renamed method. Since the name of a proof obligation is generated using a method's name, we also need to rename the partial proofs that they still match to the proof obligation. Fortunately, the renaming of methods is predictable and renaming partial proofs can be done automatically when composing proofs.

Fourth, Why requires all proof obligations to be alphabetically ordered, but proof composition can easily take the order into account when composing partial proofs.

V. Evaluation

Proof composition is a novel approach to reuse proofs in feature-oriented SPLs by composing them to larger proofs, i.e., the correctness proofs for a certain program variant. We created a small case study, which is based on our running example presented in Figure 1, to evaluate the practicability of proof composition. In Figures 2 and 3, we already presented the source code and specification for two of the five feature modules of our case study. For brevity, we omit the implementation of the other modules, but they are similar in size and complexity.

We created hand-written partial proofs for every feature module, where we omitted the proof obligation including all premises and the conclusion. Then, we generated every program variant including its specification in JML and composed the partial proofs of the features included in the configuration. We used Krakatoa and Why to generate the proof obligations into our composed proofs and let Coq prove the correctness for all program variants.

For an evaluation, we are interested how efficient the verification using proof composition is. We need to verify each program variant using Coq, similarly to variant-based verification. But, for interactive theorem proving the time needed to check whether a proof is correct is magnitudes smaller than the time needed to write the proof steps from scratch. Additionally, checking whether a proof is correct can be done automatically by a machine, while proof writing needs a human being. Hence, we can neglect the effort to check correctness of proofs and focus on proof writing.

We compare the effort of verifying our example SPL using variant-based and feature-based verification by proof composition. The effort for proof writing is related to the number of proof steps, so we introduce the measure *lines of proof* (LOP) that is similar to the measure lines of code.[2] LOP counts the number of proof steps, e.g., the LOP for Figure 6 and Figure 7 are 5 and 11, respectively.

Figure 8 shows the LOP of all partial proofs for each feature module, which is the LOP we wrote by hand for every feature. Second, we considered all generated program variants and measured the LOP for every feature, which is equivalent to the LOP we would need to write using a variant-based verification.

The variant-based LOP for feature *BankAccount* is much higher than the feature-based LOP, because the feature is included in all program variants and the partial proof is composed into the correctness proofs for all these variants. Similarly, since the other features are included in several program variants, the once hand-written proof for a feature is generated for several correctness proofs.

It is not always possible to generate the proofs for a program variant only from the partial proofs for each feature. For example, one feature may introduce an invariant and the other feature introduces a new method. Then, we get an additional proof obligation, if both features are part

[2] Note that writing a certain amount of proof lines is usually more effort than writing the same amount of code lines.

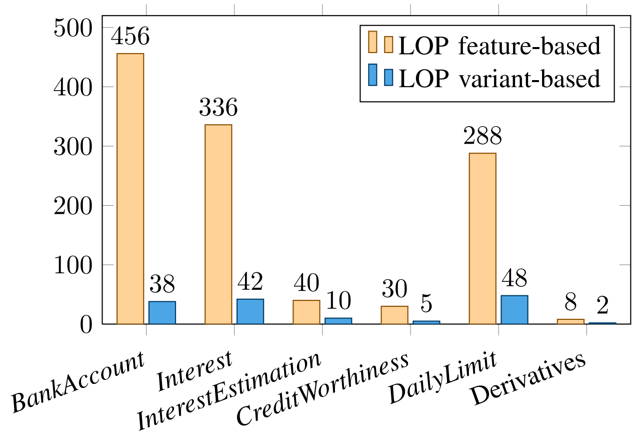

Figure 8. Variant-based vs. feature-based verfication: lines of proof (LOP)

of the same variant. This problem of structured feature interactions is well known in FOP. Feature interactions can easily be detected, since Why generates a proof obligation for which no proof exists in all variants containing the feature causing the feature interaction. Our solution is to create derivatives [13], i.e., small modules that are included, if and only if two or more other feature modules are composed. In our example, we created a derivative for the features *Interest* and *DailyLimit*, which is used to compose the proofs of four program variants.

As a result, we can sum up the proof steps over all features. The LOP using variant-based verification is 1150 steps in total. The LOP using feature-based verification is only 143. Hence, by applying proof composition, we reduced the number of hand-written proof steps by approximately 88%.

VI. Discussion

Using proof composition, we were able to save time in verifying the correctness of all program variants in our case study. In order to generalize these results, further case studies are needed. Case studies may evaluate the potential of proof composition for other domains or larger code bases. Especially, it is unclear whether proof composition can generate proofs automatically for all program variants in practice. Problems can arise with feature interactions, i.e., we need to prove something only if two features are contained in the same configuration.

Our approach is compatible with languages and tools that use superimposition [14]. This raises the question of whether proof composition can be used with other SPL implementation techniques such as delta-oriented programming (DOP) [15] or preprocessors [16]. For DOP, we would further need to be able to remove proofs or parts of proofs, e.g., if we remove a field including its class invariant. This raises the question of how to refer to parts of proofs. Proof composition will be more complicated for DOP. When implementing SPLs using preprocessors, we could imagine that preprocessor macros are used in proof documents as well.

We discovered several technical problems using Krakatoa and Why, but we assume that similar problems will arise for other verification tools. The reason is, that we need to refer to assignments, expressions, and instances in proofs for which names have to be generated. These generated names may change when adding new members to a class.

VII. Related Work

Batory and Börger [17] show how to modularize the proof that a given Java interpreter is equivalent to the JVM interpreter for Java 1.0. They modularize the Java grammar, theorems about correctness, and natural language proofs into features. In contrast, we focus on (a) Java programs instead of Java grammars, (b) allow user-defined specifications from which theorems are generated, and (c) compose machine-readable correctness proofs that can be verified by a proof assistant.

The analysis of SPLs is an active research topic. Feature model analysis [18] aims at finding inconsistencies in feature models, e.g., whether the feature model constraints are at all satisfiable. Feature interaction detection determines if a combination of features causes unwanted or unexpected behavior [10]. Single features are expressed in a formal specification language, e.g., see [11], and all pairs of potentially interacting features are checked.

In most approaches that apply model checking to SPLs, existing analysis techniques are extended to deal with optional behavior. For instance, in [19], modal transition systems are extended by variability operators from deontic logic. In [20], the process calculus CCS is extended with a variant operator to represent a family of processes. In [21], transitions of I/O-automata are related to variants. In [22], product families are modeled by transition systems where transitions are labeled with features to compute state reachability modulo a set of features. While these approaches focus on all program variants at the same time during domain engineering, there are also approaches generating constraints that need to be checked for every variant in application engineering: In [3], [4], a feature-based model checking technique is proposed relying on generated assumptions for composed features.

Apart from [17], [12], deductive verification of product lines is not yet widely considered. In [23], a correctness-by-construction approach for product lines is proposed where product features are successively refined as Event-B models. By refinement proofs, it is ensured that the properties of all features are preserved.

Type checking SPLs has similar challenges as the verification of SPLs. It is not feasible and very redundant to type check every program variant separately. Product-line–aware type systems were proposed for feature-oriented [24], [25] and annotation-based product lines [26], [27]. The idea is to type check all valid combinations of features at the same time. Given a type-safe SPL, every program variant that can be generated is type-safe. For delta-oriented product lines, a compositional type system exists, which, however, requires to check an abstraction of each program variant [28].

VIII. Conclusion and Future Work

The verification of each program variant of an SPL from scratch is a highly redundant task. We propose proof composition to generate the correctness proof for every program variant along with the source code. We compose proofs from partial proofs of the features involved.

We presented a case study of bank account programs for which we wrote partial proofs for every feature. Then, we composed the partial proofs to correctness proof for every program variant. These proofs were verified using the Coq proof assistant. Our evaluation showed that proof composition could reduce the effort of proof writing by 88%.

Future work should evaluate proof composition using larger case studies, with other tools for proof obligation generation, and other SPL implementation techniques such as preprocessors.

Acknowledgment

Apel's research was supported by the German DFG grants AP 206/2-1 and AP 206/4-1.

References

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005.

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

[3] H. C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification," in *ASE*. IEEE Computer Society, 2002, pp. 195–204.

[4] K. Fisler and S. Krishnamurthi, "Modular Verification of Collaboration-based Software Designs," in *ESECFSE*. ACM, 2001, pp. 152–163.

[5] G. T. Leavens and Y. Cheon, "Design by Contract with JML," 2005. [Online]. Available: http://www.jmlspecs.org/jmldbc.pdf

[6] Why Development Team, "Why: A Software Verification Platform," Website, available online at http://why.lri.fr/; visited on December 16th, 2010.

[7] Coq Development Team, *The Coq Proof Assistant Reference Manual*, LogiCal Project, 2010, version 8.3.

[8] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[9] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *ECOOP*, ser. LNCS, vol. 1241. Springer, 1997, pp. 419–443.

[10] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[11] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting Dependences and Interactions in Feature-Oriented Design," in *ISSRE*. IEEE Computer Society, 2010, pp. 161–170.

[12] D. Bruns, V. Klebanov, and I. Schaefer, "Verification of Software Product Lines: Reducing the Effort with Delta-oriented Slicing and Proof Reuse," in *FoVeOOS*, ser. LNCS, vol. 6528. Springer, 2010, pp. 61–75.

[13] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *JOT*, vol. 8, no. 5, pp. 49–84, 2009.

[14] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," in *ICSE*. IEEE Computer Society, 2009, pp. 221–231.

[15] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," in *SPLC*, ser. LNCS, vol. 6287. Springer, 2010, pp. 77–91.

[16] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *ICSE*. IEEE Computer Society, 2010, pp. 105–114.

[17] D. Batory and E. Börger, "Modularizing Theorems for Software Product Lines: The Jbook Case Study," *J.UCS*, vol. 14, no. 12, pp. 2059–2082, 2008.

[18] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: a Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[19] A. Fantechi and S. Gnesi, "Formal Modeling for Product Families Engineering," in *SPLC*. IEEE Computer Society, 2008, pp. 193–202.

[20] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and Model Checking Software Product Lines," in *FMOODS*. Springer, 2008, pp. 113–131.

[21] K. Lauenroth, K. Pohl, and S. Toehning, "Model Checking of Domain Artifacts in Product Line Engineering," in *ASE*. IEEE Computer Society, 2009, pp. 269–280.

[22] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines," in *ICSE*. ACM, 2010, pp. 335–344.

[23] M. Poppleton, "Towards Feature-Oriented Specification and Development with Event-B," in *REFSQ*, ser. LNCS, vol. 4542. Springer, 2007, pp. 367–381.

[24] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines," *ASE*, vol. 17, no. 3, pp. 251–300, 2010.

[25] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," in *GPCE*. ACM, 2007, pp. 95–104.

[26] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type Checking Annotation-Based Product Lines," *TOSEM*, 2011, to appear.

[27] T. Thüm, "A Machine-Checked Proof for a Product-Line–Aware Type System," Master's thesis, University of Magdeburg, Germany, 2010.

[28] I. Schaefer, L. Bettini, and F. Damiani, "Compositional Type-Checking for Delta-Oriented Programming," in *AOSD*. ACM, 2011, To appear.