Partial Proofs to Optimize Deductive Verification of Feature-Oriented Software Product Lines

Maximilian Kodetzki
Karlsruhe Institute of Technology
Karlsruhe, Germany
maximilian.kodetzki@kit.edu

Tabea Bordis
Karlsruhe Institute of Technology
Karlsruhe, Germany
tabea.bordis@kit.edu

Tobias Runge
Karlsruhe Institute of Technology
Karlsruhe, Germany
tobias.runge@kit.edu

Ina Schaefer
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

ABSTRACT

Software product lines (SPLs) are a technique to efficiently develop families of software products. Code is implemented in functional features which are composed to individual software variants. SPLs are oftentimes used in safety-critical systems, which is why functional correctness is more important than ever. As an advanced approach, deductive verification offers the possibility to verify the behaviour of software against a formal specification. When deductive verification is applied for SPLs, it meets the challenges of an SPLs variability. Since most verification approaches do not scale for variant-rich product lines, we take up existing approaches of reuse of proof parts to develop our concept of partial proofs. We split proofs into a feature-specific and a product-specific part. The feature-specific part is only proven once for all products enabling advanced proof reuse. We implement our concept of partial proofs in the tool VarCorC and evaluate it on three case studies. We found that both the number of proof steps and the verification time can be reduced by using partial proofs. Further, we determine a trend of increasing improvements of verification costs for large-scale SPLs.

KEYWORDS

software product lines, deductive verification, formal methods

1 INTRODUCTION

Functional correctness of software is a major concern, especially for safety-critical systems, where bugs can have severe consequences [29]. Usually, testing is the first choice to check the functional correctness of code. However, testing is often not sufficient as it cannot show the absence of errors [24]. One advanced option to guarantee behavioural correctness is *deductive verification* [15, 17, 18, 45]. Deductive verification is a formal method used to prove that a program satisfies a specified behaviour.

Deductive verification can also be used to verify the correctness of configurable systems [26, 44] such as *software product lines* (SPLs) [33]. SPLs aim to create families of related software products by using variable code structures. A family is defined with *features* which implement a certain functionality or behaviour. *Feature models* are used to organize the relationships and dependencies of features in a tree pattern [33]. The selection of a certain set of features is called *feature configuration* and leads to an individual software product [20]. Deductive verification can be used to guarantee the functional correctness of SPLs using formal specifications. With *feature-based specifications* [3], the code implementing a feature is specified with individual contracts consisting of a pre- and a postcondition defined in first-order logic. *Product-based verification* [41] verifies the correctness of every software product individually with respect to its specification. For that, the specification of an individual software product is built by combining the specifications of all features that implement the desired product.

Due to the exponential explosion of the number of products in a product line, product-based verification does not scale as the verification costs increase rapidly [23, 41]. Existing work on reducing the verification costs for SPLs split product-based proofs into multiple parts to enable reuse of common proof parts [12, 25, 26]. The existing approaches split the verification at method-call level as method-calls can implement variability. However, variability can also be present in specifications where variability is defined using *dependencies* on the software product considered at verification time [23]. To the best of our knowledge, dependencies in the specification have never been considered as a splitting point for proofs although they offer further possibilities to reduce the verification costs of product-based verification.

In this paper, we address this research gap and propose our concept of partial proofs to split the verification in two parts with respect to the corresponding formal specification. We divide proofs into a feature-specific part and a product-specific part. The feature-specific part does not depend on a certain software product. Thus, it is not necessary to execute this part for every software product individually. Instead, the feature-specific part can be used as a basis to continue the product-specific proof. With that, we aim to reduce the verification time and proof steps needed to verify SPLs in contrast to existing product-based verification.

For tool support and to evaluate our approach, we implement partial proofs in the tool VarCorC [9, 35]. VarCorC supports the *Correctness-by-Construction*-based [24] implementation and verification of SPLs. The correctness of an SPL is verified using deductive verification in the tool *KeY* [1]. We evaluate partial proofs regarding feasibility and whether they can reduce the verification costs of SPLs in contrast to existing product-based verification. We find significant improvements in the number of proof steps and determine a trend of decreasing verification times for variant-rich SPLs.

In summary, we make the following contributions:

- We propose partial proofs to split the verification of SPLs into a feature-specific and a product-specific part. The feature-specific part is only executed once for all software products. It is used as a basis for product-specific proofs.
- We provide tool support for partial proofs in the tool VarCorC.
- We evaluate partial proofs regarding feasibility and improvement of the verification time and proof steps needed for the verification of SPLs in contrast to existing product-based verification using three case studies.

2 BACKGROUND & MOTIVATING EXAMPLE

In this section, we present background on SPLs and introduce the *IntegerList* product line which we use throughout this paper to illustrate concepts of SPLs as well as our concept of partial proofs.

2.1 Software Product Lines

Software product line engineering is a technique to implement a set of software products variably. Depending on the desired functionality, the user can add *features* which implement certain characteristics and behaviours to a common code base [2]. The dependencies of features are modelled in *feature models* which are organized in a tree structure. The selection of features results in a *feature configuration* [5] which represents an individual software product. In Figure 1, we present the feature model of the SPL IntegerList [38]. The product line implements a list to which integers can be added. In certain configurations, the list is sorted increasingly or decreasingly. In this product line, the root feature *IntegerList* is abstract and does not contain an implementation. Abstract features are used for organizational purposes. All other features are concrete features and implement functionality. Feature *Base* is the only mandatory feature. Features *Limited* and *Sorted* are optional. Features *Increasing* and *Decreasing* are organized in an *alternative*-group, i.e., exactly one of them has to be selected as soon as the parent-feature *Sorted* is selected. Further concepts are *and*-groups (all children have to be selected) and *or*-groups (at least one child has to be selected).

Feature-Oriented Programming. To implement software families by reusing code, we use the principles of *feature-oriented programming* (FOP) [34]. In FOP, a *feature module* [2, 5] is implemented for each concrete feature of an SPL which contains classes, fields, and methods implementing the desired functionality of the corresponding feature. To receive a certain software product, feature modules are composed regarding the considered feature configuration. As a class or method can be implemented in more than one feature module, the availability of classes and methods has to be

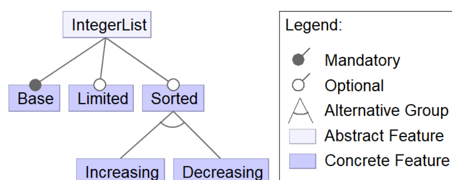


Figure 1: Feature model of the *IntegerList* product line [30].

ensured. Specific for FOP is the keyword `original`. It can be used in the implementation of a method to call the implementation of a previous feature. Principally, an original-call behaves similar to super-calls in Java, but the binding depends on the selected features. The order in which the implementations are called depends on the *feature order* which is usually defined together with the feature model [7, 34].

2.2 Software Product Line Specification

In the literature, there are various techniques to specify SPLs. Thüm et al. [41] defined a set of specification techniques after analyzing more than 100 research articles concerning the specification of SPLs. In this paper, we focus on *feature-based specifications* [3]. They can be used to define the functionality of a single feature without making explicit references to other features. For that, every method is specified individually. This results in high maintainability. Changes do not have to be propagated through all products, as the specifications are reused across the products of an SPL. For that, the keyword `\original` can be used in specifications. It references the specification of the same method from another feature.

In Figure 2, we present the implementations of the features *Base* and *Sorted* of the IntegerList product line to facilitate feature-based specifications and the keyword `\original`. The specifications of the methods are listed as JML-annotations. Feature *Base* consist of a method `push` which adds a given integer `newTop` to array `data`. The specification consists of a precondition (indicated by the keyword `requires`) and a postcondition (indicated by the keyword `ensures`). As there are no conditions, the precondition is `true` and always satisfied. For the postcondition, we use two predicates stating that (1) array `data` contains the new element and (2) array `data` contains all elements that have been in the array prior the execution of the method.[1] Feature *Sorted* also contains a method `push`. It extends the functionality of feature *Base* by sorting array `data` after adding a new element. Depending on whether feature *Increasing* or feature *Decreasing* is selected, array `data` is sorted increasingly or decreasingly calling an implementation of method `sort`. In the implementation, we use an original-call to call another implementation of this method, e.g., the implementation from feature *Base*. In general, the called method depends on the selected features and the feature order. The variability implemented with the original-call is defined in the specification of method `push` from feature *Sorted*. We use the keyword `\original` in the pre- and postcondition to refer to the conditions of the method that was called in the implementation. Further, we require array `data` to be sorted prior and after the execution of the method. When using the keyword `\original`, we have to compose the specifications of multiple implementations to build the specification of a certain product. In general, the composition process can be handled in different ways [42]. In this paper, we use the composition technique *explicit contract refinement* where the keyword is replaced by the original specification directly [39].

2.3 Software Product Line Verification

An SPL can be verified using different approaches [41]. Besides *type checking* [21] and *model checking* [16], SPLs can be verified

[1]The keyword `\old` in the postcondition references the value of the provided variable prior the execution of the method.

```
feature: Base
public int[] data;
/*@
@ requires true;
@ ensures containsNewTop(data, newTop) &&
@         containsOldElements(data, \old(data));
@*/
public void push(int newTop) {
    int i;
    int[] tmp;
    tmp = new int[data.length+1];
    tmp[tmp.length-1] = newTop;
    i = 0;
    while (i < data.length) {
        tmp[i] = data[i];
        i++;
    }
    data = tmp;
}
```

```
feature: Sorted
/*@
@ requires \original && sorted(data);
@ ensures \original && sorted(data);
@*/
public void push(int newTop) {
    original(newTop);
    sort();
}
```

Figure 2: Implementation of features *Base* and *Sorted* of the IntegerList product line including JML-specifications [30].

with respect to a formal specification using *symbolic execution* and deductive verification [12, 46]. In general, we are able to define a *Hoare-triple* [4, 19] *{P} C {Q}* with a precondition *P*, some code *C*, and a postcondition *Q* for every implemented method. The triple can be understood in such a way that if the precondition is fulfilled prior the execution of the code, the method will terminate and the postcondition will be satisfied afterwards [19, 27]. A Hoare-triple can be translated to a *dynamic logic*-formula in the form of $P \rightarrow [C]\ Q$ [12, 28]. With that, we are able to use the proof rules of the *sequent calculus* to verify the correctness of a method. The set of applicable rules is extended by *JavaDL* [1] to support the verification of Java-programs as well as individually defined proof rules. The process of applying different proof rules can be visualized in a proof tree which is to be read from the bottom to the top. Only if all branches of a proof tree can be closed, the verification is successful.

In Figure 3, we show an exemplary proof tree for method `push` from feature *Sorted* of the IntegerList product line for an exemplary configuration. As proofs usually get very large, we only show an excerpt of the proof tree. At the very bottom, we start with a dynamic logic-formula representing the Hoare-triple of method `push`. The formula is defined with respect to the code and specification shown in Figure 2. It is extended by a set of constraints to ensure that array `data` is initialized. Based on this formula, the proof rules mentioned in the previous paragraph are applied. For example, proof rule *notLeft* is applied which shifts the negated constraint *data != null* to the succedent of the implication [1]. Afterwards, proof rule *andLeft* is applied which splits the operands of the and-clause of the precondition in multiple individual preconditions [1]. Further proof rules are applied until all goals are closed. In the course of the proof, we also resolve all dependencies (i.e., the keyword `\original`). The inserted definitions depend on a certain configuration and vary for each software product. Thus, the proof has to be executed individually for each software product. This technique is called *product-based verification* and is the most intuitive verification technique for SPLs [41]. However, product-based verification does not scale for SPLs implementing a high number of software products due to the exponentially exploding verification effort.

When analyzing the proof tree presented in Figure 3, we note that the application of the first proof rules does not depend on a particular configuration as no dependencies have been resolved. For the given proof, the keyword `\original` is the only dependency, but is not resolved in the first given proof steps and likely not in a few more. As we do not resolve dependencies up to a certain point in the proof, this set of proof rules is the same for every proof and is, thus, executed multiple times unnecessarily. This causes avoidable verification costs.

3 STATE OF THE ART

Using product-based verification, every product of an SPL is verified individually. This does not scale for variant-rich SPLs. Using *family-based verification* [10, 43], one superimposed meta-product is generated and verified. This meta-product contains the code and specification of all features of an SPL and is able to simulate every software variant. The successful verification of the meta-product implies the correctness of every software product regarding the individual specification. Thüm et al. [43] presented this approach for feature-oriented SPLs implemented in Java and specified using JML. The authors could save 85% of the verification time for the considered case study in contrast to product-based verification. However, family-based verification has the major disadvantage that with every change in code or specification, the meta-product has to be re-generated and, thus, also re-verified [41].

Feature-based verification [40] is a modular approach to verify the correctness of SPLs. Using FOP, every feature is implemented in a feature module which reduces the verification effort when functionality or specification of a feature change. Only the concerned feature module has to be proven again. Nevertheless, feature-based verification is insufficiently expressive due to the interactions between features. An implementation may call methods from other features or even depend on the behaviour of another implementation [13]. To enable a sufficient feature-based verification, Knüppel et al. [23] combined feature-based verification with family-based verification in their approach of FeFaLution. Building partial proofs for every feature and reusing them for the verification of feature interactions yields to a performant verification of a complete system. The results of FeFaLution unveil reuse potential, but struggle with a substantial overhead when verifying smaller case studies.

A compositional approach that aims to reuse code and its proofs from the beginning are *trait-based languages*. Trait-based languages have also been applied to implement SPLs [8]. A trait is a class-independent set of methods which is factored for a certain purpose but can be reused in a completely different context [37]. Damiani et al. [14] proposed a proof system for deductive verification of trait-based languages. This modularity allows flexible adaption and, thus, fine-grained reuseability.

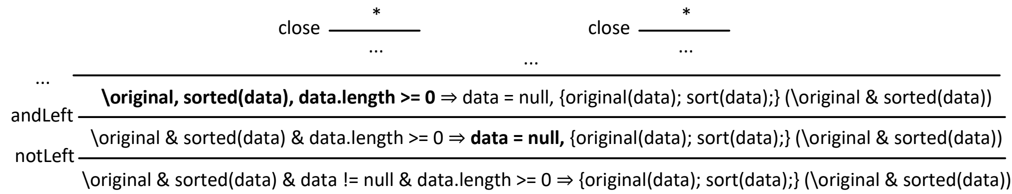


Figure 3: Excerpt of a proof tree to verify the correctness of method push from feature *Sorted* of the IntegerList product line.

A further alternative to FOP is *delta-oriented programming* [36]. A *core module* that is implementing a certain product can be modified by applying *delta modules* which change the functionality, e.g., by adding or removing fields or methods. Bruns et al. [11] proposed an approach of *delta-oriented slicing* to receive a more performant verification technique. With delta-oriented slicing, the authors determined which parts of the proof of a core module have to be re-proven for a software variant to which deltas were applied. Beckert and Klebanov [6] published an even more basic rule-by-rule reuse algorithm when repeating proofs for a corrected implementation iteratively until a successful verification is achieved. For every execution, the proof rules of the previous proof are checked and applied again, if the setting is the same or similar.

Partial proofs allow proof reuse for product-based verification [22, 25, 26]. Splitting proofs at method-call level allows to handle resolving these method-calls in different ways, depending on which method is called. Based on partial proofs, *proof plans* split verification tasks into smaller proofs [26]. These smaller proofs can be reused across the verification of different method variants. Using *proof graphs*, a wide range of possible partial proofs is uncovered, proofs can be split in an almost infinite set of small proofs. Reusing as much proof parts as possible, the verification costs for SPLs can be reduced. *Proof repositories* [12] aim to optimize the verification of method-calls and, thus, reduce the verification costs of SPLs. A proof repository contains the bindings of method-calls with a called implementation and states which bindings could already be proven successfully. This information enables a high level of reuse.

However, the existing approaches of splitting proofs focus on the variability implemented in the code by only using method-calls as variation points. To the best of our knowledge, no existing approach considers the variability defined in the specification for splitting proofs. To close this research gap, we propose our concept of splitting proofs into a feature-specific part and a product-specific part using the dependencies of specifications.

4 SOFTWARE PRODUCT LINE VERIFICATION WITH PARTIAL PROOFS

Given an SPL, the verification of its methods regarding their specifications requires high effort. In Section 2.3, we determined unnecessarily executed proof parts consisting of proof rules that are the same for all configurations when using product-based verification. The application of identical proof rules in multiple proofs causes avoidable costs. As none of the approaches presented in the previous section is able to prevent those proof rules from being applied repetitively in every proof, we take up approaches of partial proofs [23, 25, 26] and adapt them for our purposes to be able to handle product-independent proofs and reduce verification costs.

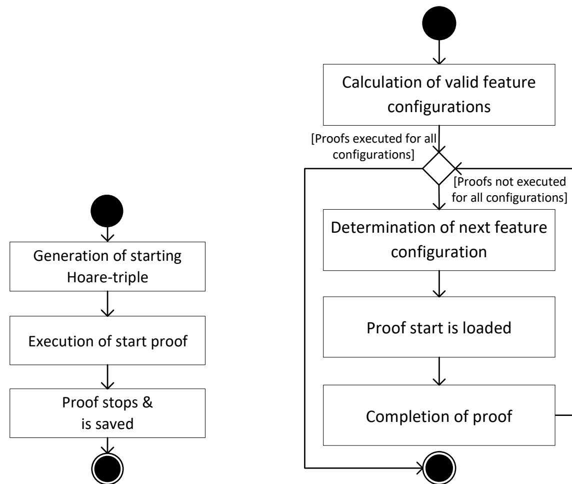
We propose our concept of *partial proofs* which split proofs in two parts. To be able to distinguish the parts, we introduce the following naming convention: the feature-specific part of a partial proof is called *proof start*. The second part is product-specific and called *proof completion*. A set of applied proof rules is identical for every proof that is conducted in the context of product-based verification of a method. This set of proof rules builds the proof start. As we do not resolve dependencies, the proof start has to be executed only once for all software products and can be reused across the verification of each software product. After performing the proof start, the proof is continued for every software product individually in proof completions. Proof completions are similar to existing product-based verification, except that we use the proof start as a basis. Dependencies are resolved, and the proof is continued until it either closes or not. In the following, we present both proof start and proof completion in detail using the example of method push from feature *Sorted* of the IntegerList product line introduced in Section 2.

4.1 Feature-Specific Proof Start

In this section, we introduce the feature-specific proof start of partial proofs in detail. In a proof start, we want to limit ourselves to the treatment of explicit clauses, i.e., the parts of a specification not containing feature dependencies to other features. With that, we ensure that proof starts do not depend on a particular software product, thus are valid for all products, and need to be performed only once for each feature.

To illustrate the concept of proof starts, in Figure 4a, we present the general verification process of proof starts. At first, the starting Hoare-triple is translated to dynamic logic (see Section 2.3). Afterwards, the proof rules of the sequent calculus are applied to verify the specification. In difference to product-based verification, we do not apply proof rules to resolve dependencies in the proof start. With that, we are not able to close the proof as it gets stuck at some point where no more proof rules can be applied. At this point, the proof start is interrupted and saved to serve as the basis for the proof completions.

We present an excerpt of the proof tree of the proof start of method push from feature *Sorted* of the IntegerList product line in Figure 5. Basically, the starting formula is the same as for existing product-based verification (cf. Figure 3). Also, the first applied proof rules are similar to product-based verification. However, we do not resolve dependencies as well as method- and original-calls that are part of the code to be verified as they may depend on a particular



(a) Verification process of feature-specific proof start. **(b) Verification process of product-specific proof completions.**

Figure 4: Verification process with partial proofs.

configuration, too. As soon as we cannot apply any more proof rules, the proof is interrupted and saved.

In Figure 5, we also show how existing product-based verification would have continued. As an example, the proof rule *originalPre* would have been applied to resolve the keyword `\original` in the premise of the implication. In this case, the precondition of the implementation from feature *Base* was used. This would be the case for a feature configuration *{Base, Sorted, Increasing}*. Finally, the proof start results in an incomplete proof that is independent of a particular software product and can thus be reused.

4.2 Product-Specific Proof Completion

In the following, we present proof completions which handle the verification of software products individually using the previously executed proof start as a basis. Reusing the proof start is already the main difference to existing product-based verification: the proof completion does not start with a formula defined in dynamic logic, but the previously executed incomplete proof tree of the proof start. To explain the details of proof completions, we show the verification process of proof completions in Figure 4b. As known from existing product-based verification, at first, all valid feature configurations with this method are calculated. Afterwards, proofs are executed for each configuration individually. For that, the previously performed proof start is continued.

In Figure 6, we show an exemplary proof completion for the feature configuration *{Base, Sorted, Increasing}*. After loading the proof start, the first applied proof rules consider dependencies, method-, or original-calls as there were no other rules that could be applied earlier. In the provided example, we apply proof rule *originalPre* to resolve the keyword `\original` of the implication's premise. In this case, the precondition of method `push` from feature *Base* is inserted. For other configurations, other preconditions may be chosen, depending on the selected features. Eventually, the inserted specification is composed of different constraints because of multiple refinements of specifications (see Section 2.2). In the context of this paper, method- and original-calls are replaced by the contract of the called method (method contracting [39]).

At the beginning of every proof completion, at least one dependency or method-call is resolved with a specification which depends on the considered software product. In general, it does not matter which dependency or method-call is resolved first. Likewise, it is usually not of major importance whether only a single or multiple dependencies, method-, or original-calls are resolved right after the start of a proof completion. Generally, this depends on the proof strategy that is used for the proof completion. After resolving one or more dependencies, the proof is conducted as in existing product-based verification. Remaining dependencies, method-, and original-calls are resolved in the course of the proof completion. Finally, every proof completion either closes or not. For each successfully closed proof, the method is proven for the considered configuration. If all proofs are closed successfully, the method is proven for all valid configurations.

4.3 Soundness of Partial Proofs

Our concept of partial proofs consisting of a feature-specific proof start and product-specific proof completions is sound by construction. Product-based verification can be represented by product-specific proof completions which are based on a single feature-specific proof start and vice versa. The soundness of partial proofs is captured in the following theorem.

THEOREM 1. *Let PL(FM) be an SPL with its features organized in a feature model FM, a set of valid feature configurations CF(PL(FM)), and a set of methods M building the implementation. Further, let PBV(m_f, c) be the product-based verification of a method m from feature f for a feature configuration c. Let PS(m_f) be a proof start of a method m from feature f and PC(m_f, c) be a proof completion for method m from feature f based on a feature configuration c. Then, for each method $m_f \in M$ with $f \in FM$, the following holds:*

$$\forall c_i \in CF(PL(FM)) : PBV(m_f, c_i) \textit{ iff } (PS(m_f) \textit{ and } PC(m_f, c_i))$$

Proof. Every product-based proof can be cut in two parts, so that the first does not depend on the considered feature configuration. Vice versa, each pair of proof start and proof completion can be combined to a product-based proof. □

4.4 Discussion

In general, the usefulness of partial proofs depends on the code and specification of a method. A method whose specification contains no or only a few dependencies, e.g., the keyword `\original`, but contains a large number of feature-specific constraints, is more likely to benefit from partial proofs than a method whose specification contains a high proportion of dependencies. This is due to the fact that in the latter case, fewer proof rules can be applied in the proof start, as it cannot proceed without resolving dependencies at an early point of time. This results in low improvements of the verification costs. The same applies for method- and original-calls. The more calls are implemented in a method, the earlier the proof start comes to an end, since product-specific continuation is required.

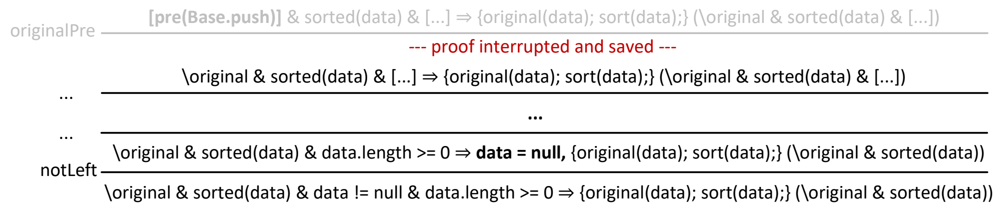


Figure 5: Excerpt of the proof tree for a proof start to verify the correctness of method push from feature *Sorted* of the IntegerList product line.

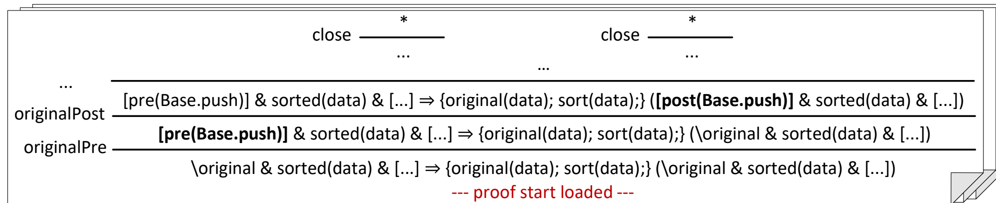


Figure 6: Excerpt of an exemplary proof completion for method push from feature *Sorted* of the IntegerList product line for the feature configuration *{Base, Sorted, Increasing}*.

On the contrary, a method that does not have any dependencies in code and specification only has to be verified one time in general. Such a method does not depend on a certain software product, performing the proof start does not result in an interruption due to missing applicable proof rules, but a closed proof. Thus, performing proof completions is not necessary.

5 TOOL SUPPORT

For the evaluation of our concept, we extend the Eclipse-plugin VarCorC [10, 35] which allows the implementation and verification of SPLs using *Correctness-by-Construction* (CbC) [24] to incrementally implement software using formal refinement rules. VarCorC offers a graphical and a textual editor to implement individual methods. Every method is specified with its own feature-based specification defined in dynamic logic for Java (JavaDL) [10] which can be verified with respect to an implementation using product-based verification and the tool KeY [1]. KeY is used to perform deductive verification automatically using the proof rules of the sequent calculus.

To support partial proofs, we split the general verification process. When triggering a proof start, we translate the starting Hoare-triple to dynamic logic and pass it to KeY as known from existing product-based verification. In addition, we pass the set of proof rules, that must not be applied by KeY. These are the rules that resolve dependencies. Further, we prohibit resolving method- and original-calls. KeY performs a proof start and returns to VarCorC, as soon as no more proof rules can be applied. This state of the proof is saved by VarCorC as an unfinished proof and is thus available for the execution of the proof completions. For that, we first calculate all valid feature configurations. A valid feature configuration corresponds to the underlying feature model and contains the method to be proven. After calculating the configurations, the proof start is loaded by VarCorC for each configuration individually and passed to KeY. On this basis, KeY executes the proof completions.

6 EVALUATION

Using VarCorC and three case studies, we evaluate our concept regarding feasibility and improvements of the verification costs of SPLs in contrast to product-based verification. For that, we define the following research questions:

RQ1: Is it possible to perform partial proofs to verify the correctness of SPLs in VarCorC?
RQ2: Do partial proofs reduce the number of proof steps needed for proofs in VarCorC?
RQ3: Do partial proofs reduce the verification time needed for proofs in VarCorC?

RQ1 addresses the feasibility of partial proofs. To answer this research question, we perform partial proofs for the methods of three case studies. For every implemented method, we execute both proof start and proof completions. After executing the proof start, we check the applied proof rules for the absence of rules resolving dependencies as they must not be resolved in the proof start (see Section 4.1). Further, we check whether all proof completions could be closed successfully.

We perform a performance evaluation of the verification of methods using partial proofs in comparison to product-based verification which is used in VarCorC so far. We take a look at the number of proof steps that have to be applied to verify a method (see *RQ2*) as well as the verification time (see *RQ3*). We execute three proofs for every method implemented: First, the proof start. Second, proof completions for all valid configurations. Third, we verify every method using product-based verification. After collecting all performance data, we perform a comparison of the proof steps and the

time spend for the verification. We compare the sum of proof start and proof completions with the data of product-based verification.

6.1 Subject Systems & Setup

To answer our research questions, we use three case studies. Besides the IntegerList product line (IL), we use the SPLs *BankAccount* (BA) and *Elevator* (E). The BankAccount product line [10, 43] implements basic functionality of a bank account such as withdrawals, transactions, limits, and checking the account balance. It was extended to support hourly limits for withdrawals. The Elevator product line [10, 32] simulates an elevator which can be moved, called from different floors, and entered by persons. In Table 1, we present general information on the structure of the case studies. Further, we present the numbers of original-calls as well as the numbers of specifications containing dependencies.

The evaluation is performed on a computer equipped with an 11th Gen. Intel Core i7-1185G7 with 3.00 GHz, 16.0 GB RAM, and Windows 10 Home (64bit). To take time variations as well as measurement inaccuracies into account, we run each proof ten times and calculate the median of the collected data for further evaluation. In total, we perform 25.389 proofs split on a set of 56 methods. For every method, we provide the sum of the proof steps and verification time of all valid configurations. Thus, the proof steps collected for proof completions of partial proofs contain the steps of the executed proof start once and the additional steps of the completions for every valid configuration.

6.2 Results and Discussion

We divide this section according to the research questions defined.

RQ1: Is it possible to perform partial proofs to verify the correctness of SPLs in VarCorC?
To evaluate our concept regarding feasibility, we verified all methods of the case studies IntegerList, BankAccount, and Elevator using partial proofs. We executed both proof start and proof completions for every method. Further, we checked whether the proof start is interrupted before any dependencies are resolved. As none of the proof starts contains proof rules resolving dependencies, we receive proofs that do not depend on a specific configuration. Resolving the dependencies in the proof completions is successful. KeY was able to close the proof completions for all configurations for all methods. As all dependencies are resolved correctly with their corresponding definition and every proof completion could be closed successfully, it is possible to perform partial proofs in VarCorC.

	IntegerList	BankAccount	Elevator
#Features	5	6	4
#Configurations	6	16	8
#Classes	1	3	4
#Methods	5	13	38
#original-calls	2	7	4
#Specifications with dependencies	5	16	9

Table 1: Overview of the case studies IntegerList, BankAccount, and Elevator product line and their characteristics.

RQ2: Do partial proofs reduce the number of proof steps needed for proofs in VarCorC?
The results of the evaluation of the proof steps is shown in Figure 7. We present the improvement achieved by using partial proofs in contrast to product-based verification grouped by feature and case study. For the number of proof steps, we receive consistent results. Except for one, the number of proof steps has decreased for all features by up to 90%. This is due to the set of proof rules that is applied only once for all product variants in the proof start. With that, we prevent these proof rules to be applied multiple times. However, we receive low improvements for some features and even a deterioration of the number of proof steps for one feature. This is due to the strategy of KeY, which forces to apply as many proof rules as possible before interrupting the proof start. This strategy may result in the application of proof rules that would not be applied at product-based verification which produces an overhead of applied rules. This problem was already discussed in literature, but is not solved so far [25, 31]. Nevertheless, the use of partial proofs overall reduces the verification costs regarding proof steps significantly.

RQ3: Do partial proofs reduce the verification time needed for proofs in VarCorC?
In *RQ3*, we considered the verification time used by KeY. In Figure 8, we show the improvement of the verification time achieved by using partial proofs in contrast to product-based verification. The results are not as clear as the results concerning proof steps. For the IntegerList, the time needed for the verification increased. Verifying the BankAccount and Elevator product lines, we receive mixed results. Thus, we cannot make a general statement regarding the improvement of the verification time.

Nevertheless, we found trends when looking at the results verifying single methods. Methods of features that have to be proven for a high number of configurations show better results than methods that have to be proven for a low number of configurations. This can be seen most clearly at the results of the BankAccount product line.

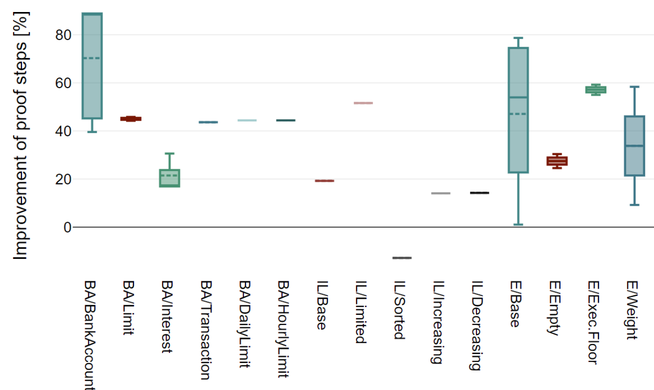


Figure 7: Improvement of the number of proof steps for partial proofs in contrast to product-based verification.

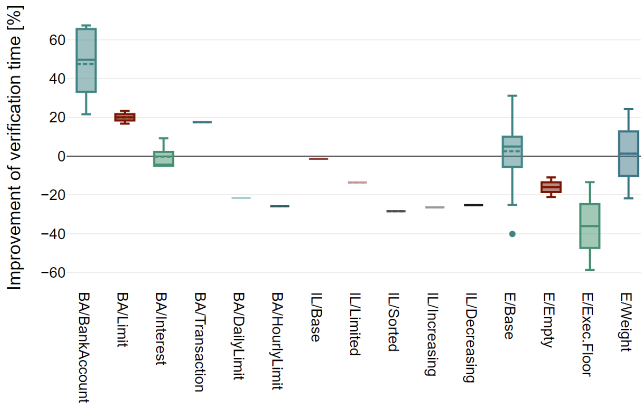


Figure 8: Improvement of verification time for partial proofs in contrast to product-based verification.

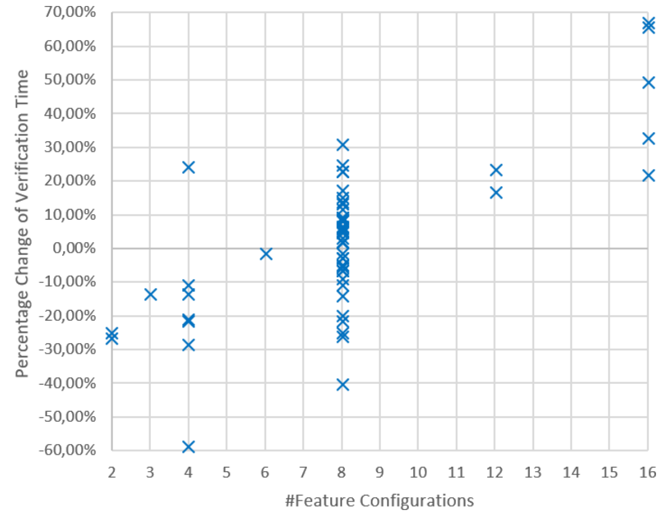


Figure 9: Improvement of verification time using partial proofs in relation to the number of valid feature configurations.

Methods from feature *BankAccount* have to be proven for 16 configurations and show the highest improvements. On the contrary, the time for verifying the methods from features *DailyLimit* and *HourlyLimit*, which only have to be proven for eight configurations, deteriorate. For the Elevator product line, methods from feature *Base* (eight valid configurations) have significantly better results than methods from the other features (four valid configurations). For the IntegerList product line, the trend is not that clear but still visible. The verification time of the method from feature *Base* (six valid configurations) has deteriorated the least. Methods that have to be verified less often, for example methods from features *Increasing* and *Decreasing* (two valid configurations each), show a much higher deterioration in verification time.

In all case studies, the main difference in verification time between the features comes from the number of configurations. For visualization, we show improvements of the verification time in relation to the number of valid configurations for the methods of all case studies in Figure 9. The more configurations have to be taken into account for the proof of a method, the higher the improvement of the verification time. We can justify this trend with the decreased proof effort. Using partial proofs, certain proof rules are applied once in the proof start. The time needed for those rules is saved for the proof completions and, thus, increases with the number of proof completions executed. Nevertheless, the verification time of methods with a low number of configurations deteriorates. This is due to the proof start, which has to be stored and loaded by KeY. This process takes much more time than loading an empty proof as it is done for product-based verification. It requires a certain number of configurations such that the time saved by applying fewer proof rules exceeds the loading time of proofs in KeY and, thus, an improvement of the verification time is recorded. For the case studies used in the evaluation, we find a break-even point of about eight configurations to receive improvements in the verification time. However, due to the limited scope of the evaluation, this number is not generalizable. We conclude that although verification time has not improved consistently across the case studies, the use of partial proofs is worthwhile in terms of verification time for product lines with many products.

6.3 Threats to Validity

Even though we have conducted and documented the evaluation with the greatest care, there are threats to the validity of the results.

Internal validity. The evaluation was executed manually in large parts. All proofs were triggered by hand but performed by KeY fully automatically. The proof steps and the verification time have been taken manually from the proof-report of KeY. All data was collected with utmost care and double-checked to exclude transmission errors. However, errors may have crept in that we did not take into account. This factor must be considered when discussing the results of the evaluation.

External validity. The case studies we used for the evaluation do not reflect the scope of SPLs used in the industry commonly. Due to limited resources and existing implemented, specified, and verified SPLs, our evaluation is not representative for large-scale SPLs. Further, the evaluation required detailed observation and monitoring which is why we performed it with smaller case studies.

7 CONCLUSION

In this paper, we presented our concept of partial proofs to improve the verification of SPLs. Splitting proofs in a feature-specific proof start and a product-specific proof completion is an alternative to existing product-based verification approaches where every product is verified individually. Using the tool VarCorC, we evaluated our concept regarding feasibility and improvements of the verification costs. We found improvements of the number of proof steps needed for partial proofs compared to product-based verification. Further, we determined a trend of improvements of the verification time for SPLs implementing a high number of products. To consolidate the trend of improvements in verification time for large-scale SPL, the evaluation could be extended by evaluating larger case studies in the future. In addition, an empirical study between partial proofs and

state of the art verification approaches, e.g. the feature-family-based verification with FeFaLution [23], could determine the significance of partial proofs.

ACKNOWLEDGMENTS

This work was partially supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF).

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmidt, and Mattias Ulbrich. 2016. *Deductive Software Verification - The KeY Book.* Springer. https://doi.org/10.1007/978-3-319-49812-6

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[3] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409. https://doi.org/10.1016/j.comnet.2013.02.025

[4] Krzysztof R. Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807. https://doi.org/10.1007/s00165-019-00501-3

[5] Don Batory. 2003. A Tutorial on Feature Oriented Programming and Product-Lines. In *25th International Conference on Software Engineering, 2003. Proceedings.* 753–754. https://doi.org/10.1109/ICSE.2003.1201271

[6] Bernhard Beckert and Vladimir Klebanov. 2004. Proof Reuse for Deductive Program Verification. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.* 77–86. https://doi.org/10.1109/SEFM.2004.1347505

[7] Fabian Benduhn. 2012. *Contract-Aware Feature Composition.* Bachelor's Thesis.

[8] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. 2015. Implementing Type-Safe Software Product Lines using Parametric Traits. *0167-6423* 97 (2015), 282–308. https://doi.org/10.1016/j.scico.2013.07.016

[9] Tabea Bordis, Maximilian Kodetzki, Tobias Runge, and Ina Schaefer. 2023. VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction. In *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops: AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26–30, 2022, Revised Selected Papers.* Springer-Verlag, Berlin, Heidelberg, 156–163. https://doi.org/10.1007/978-3-031-26236-4{_}13

[10] Tabea Bordis, Tobias Runge, David Schultz, and Ina Schaefer. 2022. Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines. *Journal of Computer Languages* 70 (2022), 101119. https://doi.org/10.1016/j.cola.2022.101119

[11] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. 2011. Verification of Software Product Lines with Delta-Oriented Slicing. In *International Conference on Formal Verification of Object-Oriented Software 2010.* Springer, Berlin, Heidelberg, 61–75. https://doi.org/10.1007/978-3-642-18070-5{_}5

[12] Richard Bubel, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. In *Transactions on Foundations for Mastering Change I.* Springer International Publishing, Cham, 130–156. https://doi.org/10.1007/978-3-319-46508-1_8

[13] Yoonsik Cheon. 2003. *A runtime assertion checker for the Java Modeling Language.* https://doi.org/10.31274/rtd-180813-9872

[14] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. 2014. Verifying Traits: A Proof System for Fine-Grained Reuse. *Formal Aspects of Computing* 26 (2014), 761–793. https://doi.org/10.1007/s00165-013-0278-3

[15] Jean-Christophe Filliâtre. 2011. Deductive software verification. *International Journal on Software Tools for Technology Transfer* 13, 5 (2011), 397–403. https://doi.org/10.1007/s10009-011-0211-0

[16] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *Formal methods for open object-based distributed systems.* Vol. 5051. 113–131. https://doi.org/10.1007/978-3-540-68863-1{_}8

[17] Rainer Hähnle and Marieke Huisman. 2017. 24 Challenges in Deductive Software Verification. In *ARCADE. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (EPiC Series in Computing, Vol. 51).* EasyChair, 37–41. https://doi.org/10.29007/j2cm

[18] Reiner Hähnle and Marieke Huisman. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science.* Springer eBooks Computer Science, Vol. 10000. Springer, Cham, 345–373. https://doi.org/10.1007/978-3-319-91908-9{_}18

[19] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[20] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study.

[21] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology* 21, 3 (2012), 1–39. https://doi.org/10.1145/2211616.2211617

[22] Vladimir Klebanov. 2007. Proof Reuse. In *Verification of Object-Oriented Software. The KeY Approach.* Springer, Berlin, Heidelberg, 507–529. https://doi.org/10.1007/978-3-540-69061-0{_}13

[23] Alexander Knüppel, Stefan Krüger, Thomas Thüm, Richard Bubel, Sebastian Krieter, Eric Bodden, and Ina Schaefer. 2020. Using Abstract Contracts for Verifying Evolving Features and Their Interactions. In *Deductive Software Verification: Future Perspectives.* Springer, Cham, 122–148. https://doi.org/10.1007/978-3-030-64354-6{_}5

[24] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-by-Construction Approach to Programming.* Springer, Berlin and Heidelberg. https://doi.org/10.1007/978-3-642-27919-5

[25] Elias Kuiter. 2020. *Proof Repositories for Correct-by-Construction Software Product Lines.* Master's Thesis. Otto-von-Guericke University Magdeburg, Magdeburg.

[26] Elias Kuiter, Alexander Knüppel, Tabea Bordis, Tobias Runge, and Ina Schaefer. 2022. Verification Strategies for Feature-Oriented Software Product Lines. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems.* ACM, New York, NY, USA. https://doi.org/10.1145/3510466.3511272

[27] Gary T. Leavens and Cheon Yoonsik. 2006. *Design by Contract with JML.*

[28] S. D. Litvintchouk and V. R. Pratt. 1977. *A Proof-Checker for Dynamic Logic.*

[29] Jing Liu, Josh Dehlinger, and Robyn Lutz. 2007. Safety analysis of software product lines using state-based modeling. *Journal of Systems and Software* 80, 11 (2007), 1879–1892. https://doi.org/10.1016/j.jss.2007.01.047

[30] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer, Cham. https://doi.org/10.1007/978-3-319-61443-4

[31] Maria Pelevina. 2015. *Realization and Extension of Abstract Operation Contracts for Program Logic.* Bachelor's Thesis.

[32] Malte Plath and Mark Ryan. 2001. Feature integration using a feature construct. *Science of Computer Programming* 41, 1 (2001), 53–84. https://doi.org/10.1016/S0167-6423(00)00018-6

[33] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering.* Springer, Berlin and Heidelberg. https://doi.org/10.1007/3-540-28901-1

[34] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming.* Springer, Berlin, Heidelberg, 419–443. https://doi.org/10.1007/BFb0053389

[35] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. In *FASE 2019: Fundamental Approaches to Software Engineering.* Springer, Cham, 25–42. https://doi.org/10.1007/978-3-030-16722-6{_}2

[36] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *SPLC 2010 – Software product lines.* 77–91. https://doi.org/10.1007/978-3-642-15579-6{_}6

[37] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003 – Object-Oriented Programming.* 248–274. https://doi.org/10.1007/978-3-540-45070-2_12

[38] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. 2011. Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report. In *Proceedings of the 15th International Software Product Line Conference, Volume 2* (Munich, Germany) *(SPLC '11).* Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/2019136.2019144

[39] Thomas Thüm. 2015. *Product-Line Specification and Verification with Feature-Oriented Contracts.* Dissertation.

[40] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. 2012. Analysis Strategies for Software Product Lines.

[41] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 1–45. https://doi.org/10.1145/2580950

[42] Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. 2019. Feature-Oriented Contract Composition. *Journal of Systems and Software* 152 (2019), 83–107. https://doi.org/10.1016/j.jss.2019.01.044

[43] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering.* ACM, New York, NY, 11–20. https://doi.org/10.1145/2371401.2371404

[44] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. 2011. Proof Composition for Deductive Verification of Software Product Lines. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops.* 270–277. https://doi.org/10.1109/ICSTW.2011.48

[45] Alan Turing. 1949. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating machines.* Mathematical Laboratory, Cambridge,

67–69.
[46] Dries Vanoverberghe, Nikolaj Bjørner, Jonathan de Halleux, Wolfram Schulte, and Nikolai Tillmann. 2008. Using Dynamic Symbolic Execution to Improve Deductive Verification. In *Model checking software*. Vol. 5156. 9–25. https://doi.org/10.1007/978-3-540-85114-1{_}4