Verification Strategies for Feature-Oriented Software Product Lines

Elias Kuiter
Otto-von-Guericke-University
Magdeburg, Germany
kuiter@ovgu.de

Alexander Knüppel
TU Braunschweig
Braunschweig, Germany
a.knueppel@tu-bs.de

Tabea Bordis
TU Braunschweig
Braunschweig, Germany
t.bordis@tu-bs.de

Tobias Runge
TU Braunschweig
Braunschweig, Germany
tobias.runge@tu-bs.de

Ina Schaefer
TU Braunschweig
Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

Highly-customizable software systems in form of software product lines are becoming increasingly relevant for safety-critical systems, in which the correctness of software is a major concern. To ensure the correct behavior of a software product line, each product can be verified in isolation—however, this strategy quickly becomes infeasible for a large number of products.

In this paper, we propose proof plans, a novel strategy for verifying feature-oriented software product lines based on partial proofs. Our technique splits the verification task into small proofs that can be reused across method variants, which gives rise to a wider spectrum of verification strategies for software product lines. We describe applications of our technique and evaluate one of them on a case study by comparing it with established verification strategies.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Formal software verification**.

KEYWORDS

Software Product Lines, Deductive Verification, Proof Reuse

ACM Reference Format:
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 (VAMOS '22), February 23–25, 2022, Florence, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3510466.3511272

1 INTRODUCTION

In the software industry of today, there is an increased demand for highly-customizable software systems [13, 29]. *Software product lines* (SPLs) are a methodology to plan, develop, and maintain such systems [3]. In an SPL, a large number of products can be derived in an automated fashion from a shared set of assets [10]. This methodology allows for mass customization of software, reduction of development costs, and sustained maintainability [10, 20, 29]. In an SPL, products are distinguished by the presence or absence of *features* that implement end-user visible behavior of the software system [3]. With an extension of OOP known as *feature-oriented programming* (FOP), such features can be developed modularly [7].

SPLs are increasingly used for safety-critical systems, where correct software behavior is a major concern [26]. In *deductive verification*, correctness of software is derived as a formal proof in a verification calculus [1]. To specify the correctness of a piece of code (e.g., a method), developers annotate methods with machine-checkable contracts, which consist of a precondition (i.e., what a method expects from its caller) and a postcondition (i.e., the program state guaranteed if the precondition is met) [27].

The deductive verification of SPLs is challenging due to the potentially exponential number of products [33, 35]. In particular, a mere *product-based strategy* (in which each product is verified separately) is only feasible for SPLs with few products [33]. To facilitate large-scale SPL verification, techniques have been developed that avoid redundant effort [5, 33, 36]. The state of the art (a *family-based strategy*) is to encode all products of an SPL into a single *metaproduct* [35] that can then be verified as a whole with an off-the-shelf verification system such as KeY [1]. The verification system can then leverage similarities between products to optimize the verification effort. However, this approach does not adapt well to changes, which may have a large impact on the metaproduct and require costly re-verification [33]. To mitigate this issue, we may alternatively split the verification task into many small *partial proofs* [8, 21], which can be reused to verify similar or evolving products. A partial proof may leave "gaps" for proof steps that concern method calls, which can later be *bound* to concrete methods. So, we can first conduct partial proofs for individual methods (leaving all gaps unbound) and then complete all proofs by successively binding method calls to concrete methods. Such a *feature-family-based strategy* can also be used in evolution scenarios, as changes to implementation artifacts only invalidate some partial proofs [24]. However, previous evaluations of partial proofs show mixed results with regard to reduction in verification effort, which suggests that partial proofs are inferior to metaproducts in this regard [8, 21].

In this paper, we propose a novel partial-proof-based technique for verifying feature-oriented SPLs. In contrast to previous work

VAMOS '22, February 23–25, 2022, Florence, Italy
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9604-2/22/02.
https://doi.org/10.1145/3510466.3511272

(e.g., abstract contracts [9, 17]), we are not concerned with *how* to conduct partial proofs, rather how to *apply* partial proofs for SPL verification. To this end, we explicitly model SPL verification strategies with *proof plans*. A proof plan is a tree structure that describes how an SPL (or part thereof) can be verified by conducting partial proofs. Analogous to query plans in database systems [12], proof plans are manipulable first-class objects that span a space of all possible verification strategies for an SPL (i.e., its *proof graph*), which may be explored to address post-hoc and evolution scenarios for SPL verification. This sets proof plans apart from previous partial-proof-based approaches, such as proof repositories [8], which ignore the possibility for different verification strategies, or Fefalution [21], which only makes limited use of partial proofs.

We describe several kinds of proof plans with varying degree of proof reuse. In particular, feature-family-based proof plans guarantee proof reuse, by which we aim to address the mixed evaluation results for partial proofs. Thus, our technique may be used to strike a balance between only conducting monolithic, product-based proofs (which prohibit proof reuse) and only conducting partial proofs (which enable proof reuse, but impose additional overhead).

In summary, we contribute the following in this paper:

- We propose a sound formalization of proof plans, a novel feature-family-based technique for verifying feature-oriented SPLs.
- We express existing techniques for SPL verification in terms of proof plans, which yields a spectrum of SPL verification strategies, and we discuss practical applications of proof plans.
- We implement proof plans in a prototype based on the KeY verification system and evaluate it on a small case study.

As proof reuse (e.g., by feature-family-based proof plans) naturally aligns with the goal of software reuse in the SPL community, our work lays the theoretic groundwork for a more comprehensive discussion on the modular verification of feature-oriented SPLs.

2 FORMAL FOUNDATIONS

We first give a brief introduction into deductive verification with partial proofs and feature-oriented software product lines.

2.1 Partial Proofs in Deductive Verification

We base our technique on *Hoare triples*, which comprise code and its specification [16]. A Hoare triple $\{P\}\ S\ \{Q\}$ consists of a precondition P, a sequence of statements S, and a postcondition Q. Pre- and postconditions are first-order logic formulas with access to program locations, while statements are given as code in some programming language (e.g., guarded command language [11]; in our examples, we use a Java-like language). Semantically, a Hoare triple $\{P\}\ S\ \{Q\}$ corresponds to the formula $P \rightarrow \langle S \rangle\, Q$ in dynamic first-order logic (DL) [14]. This formula is valid if, whenever the precondition P holds, the implementation S terminates and the postcondition Q holds afterwards (i.e., total correctness). For our technique, we need not distinguish between total and partial correctness.

Programs are structured around *methods* and their *calls*. In the following, we use the notation $\mathtt{m}(p_1, \ldots, p_n)\colon \{P\}\ S\ \{Q\}$ to refer to a method named m that takes parameters p_i. The precondition P and postcondition Q specify what the method expects when it is called and what it guarantees afterwards (i.e., the method's contract). The method's implementation is then represented as a sequence of statements S. A method m can call another method n in S, which we denote as m.n. With *Calls*(m), we refer to the set of all calls in m.

Example 1. *Consider the method* $\mathtt{ins}(A, x)$ *given by*

$$\begin{aligned} \mathtt{ins}(A, x)\colon\ & \{\mathit{Sorted}(A)\} \\ & \mathtt{original}(A, x);\ \mathtt{sort}(A) \\ & \{\mathit{Sorted}(A) \land x \in A\}, \end{aligned}$$

where we assume a helper predicate Sorted$(\cdot)$. This method inserts an element x into a sorted list A; and we want to guarantee that the resulting list is still sorted and contains x. The method ins *uses a call to* original *(a keyword in feature-oriented programming, cf. subsection 2.2) and a subsequent call to* sort *(e.g., Calls*(ins) $= \{\mathtt{ins.original}, \mathtt{ins.sort}\}$*).*

To interpret calls as in Example 1, we need *bindings* to concrete methods, so we must track which method contains which calls. We use the notation $\mathtt{m.n} \mapsto \mathtt{m'}$ to denote that a call m.n is *bound* to another method m′with a matching signature. Given a set of methods M and a set of such bindings B, each call of a method in M can be bound to none, one, or many concrete methods in M by bindings in B. When B only contains up to one binding per call, we consider B to be *deterministic*—to represent variation in SPLs, we also need nondeterministic binding sets (cf. subsection 2.2). Regarding Example 1, consider $M = \{\mathtt{ins}, \mathtt{origList}, \mathtt{origSet}, \mathtt{sort}\}$ and $B = \{\mathtt{ins.original} \mapsto \mathtt{origList}, \mathtt{ins.original} \mapsto \mathtt{origSet}, \mathtt{ins.sort} \mapsto \mathtt{sort}\}$. The call ins.original is bound to two methods (origList/origSet), which encode two variants of ins (e.g., list and set insertion). In subsection 2.2, we show how to represent SPLs, continuing Example 1.

To verify the correctness of method variants with respect to their specification, we define proofs as sequences of *proof steps*, where a proof step transforms a *proof obligation set* (i.e., a set of DL formulas) into another by binding a given call. A proof for a method $\mathtt{m}\colon \{P\}\ S\ \{Q\}$ starts with the proof obligation set $\{P \rightarrow \langle S \rangle\, Q\}$, which corresponds to the satisfaction of m's contract. In this proof obligation set, a call may be bound by replacing it with the contract of the bound method. By removing all proof obligations that become provable (i.e., deducible in a first-order logic calculus), we are left with a set of proof obligations that remain to be proven. We can repeat this process and successively bind calls, generating new (reduced) proof obligation sets. The proof is *partial* as long as there are unbound calls left. As soon as all calls are bound, the proof becomes *complete*; then it may be *closed* or not (i.e., *open*), depending on whether we succeeded to reduce the proof obligation set to the empty set $\varnothing$.

Definition 1 (Proof). *A* proof obligation set *is a set of DL formulas Φ. Let Φ be a proof obligation set and $b = \mathtt{m.n} \mapsto \mathtt{m'}$ a binding. Then $\Phi \xrightarrow{b} \Phi'$ is a* proof step, *where*

$$\Phi' := \{\phi' \mid \phi' \in \mathit{SubstContract}(\phi, \mathtt{n}, \mathtt{m'}), \mathit{NotProvable}(\phi'), \phi \in \Phi\}.$$

A proof $\bar{\Phi}$ *for a method* $\mathtt{m}\colon \{P\}\ S\ \{Q\}$ *is a sequence of proof steps* $\bar{\Phi} := \Phi_0 \xrightarrow{b_1} \ldots \xrightarrow{b_n} \Phi_n$ *that starts with the proof obligation set* $\Phi_0 = \{P \rightarrow \langle S \rangle\, Q\}$.

A proof $\bar{\Phi}$ *is* complete *when each call in Calls*(m) *is bound by b_i for some i, otherwise it is* partial. *It is* closed *when $\Phi_n = \varnothing$.*

By supplying bindings with $\xrightarrow{b}$, we enforce a separation between calls and called methods (inspired by abstract contracts [9, 17]). Complete proofs constructed with this separation are equivalent to traditional proofs [17]. However, our approach explicitly allows partial proofs, in which not all calls are bound yet. Below, we demonstrate how partial proofs can be paused and continued when bindings become available to achieve proof reuse.

In Definition 1, we omit technical details that are orthogonal to our approach: First, we do not go into detail about how to replace a call to a method `n` with the contract of method `m'`. Instead, we assume a function *SubstContract* that takes a formula ϕ, applies method contracting (e.g., with abstract contracts [17]), and returns a set of new proof obligations. When ϕ does not contain any call to `m'`, *SubstContract* returns $\{\phi\}$ unchanged. Second, we abstract from ordinary DL proof steps with the predicate *NotProvable*, which checks the DL validity of proof obligations when they have no unbound calls. In an implementation of our concept, this is where the actual verification effort lies, as a verification system (e.g., KeY [1]) must be invoked to check DL validity. Third, we make the assumption that the underlying verification calculus is sound; that is, a closed proof for a method implies that the method satisfies its contract.

In the following lemma, we observe that the order of proof steps does not affect correctness of proofs; that is, bindings commute.

Lemma 1 (Commutativity of Bindings). *Let $\bar{\Phi}$ be a proof and $\{b, b'\}$ a deterministic set of bindings such that $\bar{\Phi} \xrightarrow{b} \Phi_u \xrightarrow{b'} \Phi_v$ and $\bar{\Phi} \xrightarrow{b'} \Phi_w \xrightarrow{b} \Phi_x$ are also proofs. Then, $\Phi_v = \Phi_x$.*

Proof. This is due to *SubstContract* always being applied to disjoint subformulas representing the substituted method calls. □

In particular, Lemma 1 allows us to safely extend the proof step relation $\Phi \xrightarrow{b} \Phi'$ to sets of deterministic bindings; that is, we can bind multiple calls with $\Phi \xrightarrow{\{b_1,\dots,b_n\}} \Phi' := \Phi \xrightarrow{b_1} \dots \xrightarrow{b_n} \Phi'$.

With smart use of partial proofs, we can achieve proof reuse in Example 1. That is, we can conduct the following proofs:

$$\bar{\Phi}_u = \Phi_0 \xrightarrow{\texttt{ins.sort}\mapsto\texttt{sort}} \Phi_u$$
$$\bar{\Phi}_v = \bar{\Phi}_u \xrightarrow{\texttt{ins.original}\mapsto\texttt{origList}} \Phi_v$$
$$\bar{\Phi}_w = \bar{\Phi}_u \xrightarrow{\texttt{ins.original}\mapsto\texttt{origSet}} \Phi_w$$

First, we construct a partial proof $\bar{\Phi}_u$. Because $\bar{\Phi}_u$ may rely on the definition of `sort`, this reduces the proof obligation set $\Phi_0 = \{Sorted(A) \rightarrow \langle \texttt{original}(A,x); \texttt{sort}(A)\rangle\, Sorted(A) \wedge x \in A\}$ to $\Phi_u = \{Sorted(A) \rightarrow \langle \texttt{original}(A,x)\rangle\, x \in A\}$. To check whether both variants of `ins` are correct, we can construct two complete proofs $\bar{\Phi}_v$ and $\bar{\Phi}_w$ and determine whether both are closed. By reusing the proof $\bar{\Phi}_u$ for both variants, we achieve proof reuse.

2.2 Feature-Oriented Software Product Lines

A software product line (SPL) is a family of programs (i.e., products) constructed from reusable artifacts [10, 20, 29]. We focus on feature-oriented SPLs, in which *features* distinguish products, each of which is characterized by a *configuration* of selected features [3, 18].

$$\begin{aligned}
F &= [Base, Set, Ord] \\
C &= \{[Base], [Base, Set], [Base, Ord], [Base, Set, Ord]\} \\
M &= \{Base{::}\texttt{ins}(A,x),\ Base{::}\texttt{find}(A,x),\ Ord{::}\texttt{find}(A,x), \\
&\quad Set{::}\texttt{ins}(A,x)\colon\ \{\dots\}\ \text{if } \neg\texttt{find}(A,x) \\
&\qquad\qquad \text{then } \texttt{original}(A,x)\ \{\dots\}, \\
&\quad Ord{::}\texttt{ins}(A,x)\colon\ \{\dots\}\ \texttt{original}(A,x);\ \texttt{sort}(A)\ \{\dots\}, \\
&\quad Ord{::}\texttt{sort}(A)\colon\ \{\dots\}\ \dots\texttt{sort}(A[0, \tfrac{|A|}{2}]) \\
&\qquad\qquad \dots\texttt{sort}(A[\tfrac{|A|}{2}, |A|])\dots\ \{\dots\}\}
\end{aligned}$$

Figure 1: Example SPL for (un-)sorted sets and lists.

As a means for implementing SPLs, we use feature-oriented programming (FOP) [30, 31], in which methods are refined with a special `original` keyword [7]. With this keyword, a method in a refining feature f can invoke its *original* method in a refined feature f'; similar to how the `super` keyword invokes a method of a superclass in OOP.[1] The actual method called by `original` depends on the configuration, which gives rise to variability.

We formally define feature-oriented SPLs. In the following, we use brackets ([...]) to indicate totally ordered sets (where $\subseteq$ preserves the order) and $\mathcal{P}(\cdot)$ for the power set.

Definition 2 (SPL). *An SPL $\ell = (F, C, M)$ has*

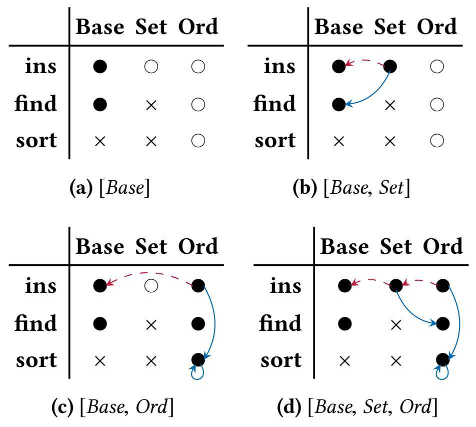
- *an ordered set of* features $F = [f_1, \dots, f_n]$,
- *a set of* configurations $C \subseteq \mathcal{P}(F)$, *each being an ordered set of selected features, and*
- *a set of* methods M, *each of which is annotated with a containing feature (e.g., a method $f{::}\texttt{m} \in M$ is named* `m` *and belongs to f).*

We refer to all calls in an SPL as $Calls(\ell) := \bigcup_{f::\texttt{m} \in M} Calls(f{::}\texttt{m})$.

A method `m` may be implemented in several features (e.g., $f_1{::}\texttt{m}$, $f_2{::}\texttt{m}$, ...), but each method may only be implemented once per feature. We omit the feature (and write `m` only) when irrelevant. Features (and, thus, configurations) have a defined order, which determines the meaning of `original` calls; similar to how a class hierarchy determines the meaning of `super`. However, unlike `super` in OOP, the target of an `original` call is determined statically.

In Figure 1, we show an example SPL that represents a small family of list data structures. This SPL consists of three features *Base*, *Set*, and *Ord* implemented by methods in M (for brevity, we omit all method contracts). The first feature *Base* implements basic list algorithms, namely *Base*::`ins` (which inserts an element x into a list A) and *Base*::`find` (which returns whether an element x occurs at least once in a list A; e.g., by linear search). The feature *Set* refines *Base*::`ins` as it additionally ensures uniqueness of elements by calling a method named `find`, which may refer to several concrete methods (i.e., *Base*::`find` or *Ord*::`find`). Finally, the feature *Ord* represents a sorted list by implementing a method *Ord*::`sort`, which sorts a list A recursively. To insert a new element into a sorted list, *Ord* invokes the original `ins` method and calls `sort` afterwards (cf. Example 1). Also, *Ord* refines the method `find` to implement an optimized search algorithm (e.g., binary search). Then, the SPL

[1]For simplicity, we disregard the possibility for `original` occurrences in contracts. However, such explicit contract refinement can be supported as well [24].



(a) [*Base*] **(b)** [*Base*, *Set*] **(c)** [*Base*, *Ord*] **(d)** [*Base*, *Set*, *Ord*]

Figure 2: Collaboration diagrams for the SPL from Figure 1.

in Figure 1 consists of four products represented by configurations in C: Unsorted lists ([*Base*]), unsorted sets ([*Base*, *Set*]), sorted lists ([*Base*, *Ord*]), and sorted sets ([*Base*, *Set*, *Ord*]).

Given an SPL as in Figure 1, we can derive a product for a given configuration. To this end, we must specify which methods are included and which call is bound to which method in that product. In the following, $\subseteq$ and $=$ must respect the order of their operands.

Definition 3 (SPL Products). *Let $\ell = (F, C, M)$ be an SPL and $c \in C$ a configuration. The ordered set $c_{\mathtt{m}}:=[f \in c \mid f::\mathtt{m} \in M]$ restricts c to all features that implement* m. *The* methods *in c are given by $M(c) = \{f::\mathtt{m} \in M \mid f \in c\}$, while the* bindings *in c are given by*

$$\begin{aligned}B(c):=\{&f::\mathtt{m.original} \mapsto f'::\mathtt{m} \mid \\ &f::\mathtt{m.original} \in \mathit{Calls}(\ell), [f', f] \subseteq c_{\mathtt{m}}\} \\ \cup\{&f::\mathtt{m.n} \mapsto f'::\mathtt{n} \mid \\ &f::\mathtt{m.n} \in \mathit{Calls}(\ell), \mathtt{n} \neq \mathtt{original}, c_{\mathtt{n}} = [\ldots, f']\}\end{aligned}$$

The set of all bindings in the SPL ℓ is then defined as $B(\ell):=\bigcup_{c \in C} B(c)$.

In Figure 2, we exemplify this definition with collaboration diagrams [31] for all configurations of the example SPL in Figure 1. Columns correspond to features, rows to method names, and cells to methods; so each cell is a method implemented in its column's feature (× marks unimplemented methods). A cell is marked with ● when its method is included in the product (or ○ otherwise). A product includes all methods implemented in some selected feature.

Each arrow in Figure 2 is a binding from some call to some method in the derived product. For example, all `sort` calls (solid arrows, including the loops) are bound to *Ord*::`sort`, the only method in the `sort` row. In contrast, `original` calls (dashed arrows) are always bound to a method of the same name whose feature precedes the current feature in F. So, dashed arrows always point to some cell to the left in the same row, depending on the configuration. For example, `original` in *Ord*::`ins` is bound to *Set*::`ins` in the configuration [*Base*, *Set*, *Ord*] and to *Base*::`ins` in [*Base*, *Ord*].

With this notion of product derivation, we formally define when we consider an SPL to be correct.

Definition 4 (SPL Correctness). *An SPL $\ell = (F, C, M)$ is* correct *(denoted by $\ell^{\checkmark}$) when for all configurations $c \in C$ and methods $\mathtt{m}\colon \{P\}\ S\ \{Q\} \in M(c)$, the proof $\{P \longrightarrow \langle S \rangle\ Q\} \xrightarrow{B(c)} \varnothing$ is closed.*

That is, an SPL ℓ is correct if the proofs for every method in every configuration c (where all calls are bound according to the deterministic set of bindings $B(c)$), are closed. Assuming that ℓ contains no dangling method references, these proofs are complete and thus equivalent to traditional correctness semantics [17].

Definition 4 describes a product-based verification strategy, where each product is verified in isolation [33]. However, such a strategy is infeasible for large configuration spaces. Even for our small example SPL, already 16 proofs are necessary (i.e., the number of ● cells), all of which have potential overlap with at least one other proof, as each method occurs in at least two configurations. This shows the potential for proof reuse even in small SPLs.

3 PROOF PLANS

For Example 1, we showed how reusing partial proofs may lead to a reduction in verification effort. In such a small example, it is easy to manually locate the potential for reuse and determine which partial proofs should be conducted. For entire SPLs, this procedure is less obvious, as the call structure (and, thus, collaboration diagrams) may be complex, so there may be many strategies for conducting partial proofs that differ in their verification effort.

To systematically leverage commonalities between proofs, we propose to track all possible proofs that may be conducted in a *proof graph* spanning the entire space of possibilities for verifying an SPL. Each node in a proof graph (denoted by $\mathtt{m}[B]$) represents all possible proofs for some method m, in which calls are bound according to a set of bindings B. Each edge from one node to another then shows how to bind some new calls in any of these possible proofs, which leads to a new set of possible proofs that include the new bindings.

The resulting graph uncovers a rich structure of partial proofs, which we exploit to achieve proof reuse in an SPL.

Definition 5 (Proof Graph). *Let M be a set of methods. A* proof graph *$g = (N, E)$ is a directed acyclic graph with*

- *a set of* nodes *$N \subseteq \{\mathtt{m}[B] \mid \mathtt{m} \in M, B$ is a deterministic set of bindings$\}$, each of which represents a method* m *whose calls are bound deterministically by the bindings given in B and*
- *a set of* edges *$E \subseteq \{(\mathtt{m}[B], \mathtt{m}[B']) \in N \times N \mid B \subset B'\}$ between those nodes, each adding some new bindings $B' \setminus B$ to a node.*

A source (sink) node *in g has no incoming (outgoing) edges in E. We refer to source (sink) nodes with $\mathit{Sources}(g) \subseteq N$ ($\mathit{Sinks}(g) \subseteq N$).*

In Figure 3, we show part of the proof graph for the example SPL from Figure 1 (below, we show how exactly to construct such a proof graph). In this proof graph, each method $\mathtt{m} \in M$ has one connected component that starts with the method's corresponding source node $\mathtt{m}[\varnothing]$. This source node branches out into several possible paths whose branching structure depends on the bindings of the calls in m. All paths then lead to some sink node (marked gray), which represents a particular variant of the method m.

Given a proof graph, we define its semantics (i.e., the correctness of all method variants it represents) by conducting proofs for each node as follows: That is, we label each node $\mathtt{m}[B]$ with a set of proofs for $\mathtt{m}$ that bind calls as given by B. Each of these proofs represents one particular order in which the bindings in B may be added. Consequently, for all source nodes, there is only one such proof, which consists of the initial proof obligation (and possibly, some initial bindings as well). For all non-source nodes $\mathtt{m}[B']$, we look up predecessor nodes $\mathtt{m}[B]$ with edges incoming to $\mathtt{m}[B']$; that is, nodes with proofs that can be continued as proofs for $\mathtt{m}[B']$. We then continue all these proofs by binding the missing calls given by $B' \setminus B$, arriving at a new set of proofs for $\mathtt{m}[B']$. Thus, we conduct longer (and more complete) proofs by following the edges in the proof graph. The sink nodes in the proof graph are then labeled with sets of complete proofs, which must all be closed for the proof graph to be correct. Intuitively, as each sink node corresponds to one method variant in an SPL, the correctness of proof graphs then corresponds to SPL correctness.

Definition 6 (Proof Graph Semantics). *Let $g = (N, E)$ be a proof graph. We label each node $\mathtt{m}[B] \in N$ with a set $Proofs(\mathtt{m}[B])$:*

- *For any source node $\mathtt{m}[B] \in Sources(g)$ and any method $\mathtt{m}\colon \{P\}\ S\ \{Q\}$, let $Proofs(\mathtt{m}[B]) {:=} \{\{P \longrightarrow \langle S \rangle\, Q\} \xrightarrow{B} \Phi\}$, while*
- *for any non-source node $\mathtt{m}[B'] \in N \setminus Sources(g)$, let $Proofs(\mathtt{m}[B'])$ $:=\{\bar{\Phi} \xrightarrow{B' \setminus B} \Phi \mid (\mathtt{m}[B], \mathtt{m}[B']) \in E, \bar{\Phi} \in Proofs(\mathtt{m}[B])\}$,*

where Φ is the new proof obligation set resulting from applying the bindings B and $B' \setminus B$ according to Definition 1.

The proof graph g is correct *(denoted by $g^{\checkmark}$) when for all sink nodes $\mathtt{m}[B] \in Sinks(g)$ and all proofs $\bar{\Phi} \in Proofs(\mathtt{m}[B])$, $\bar{\Phi}$ is closed.*

In Definition 6, our use of proofs is justified by the commutativity of bindings (cf. Lemma 1) and because all involved sets of bindings are deterministic. However, as there may be many incoming edges towards a node, $Proofs(\mathtt{m}[B])$ may be a set of many proofs. Thus, we cannot uniquely identify a single proof to conduct for each node.

We address this issue by introducing *proof plans*: A proof plan is a subgraph of a proof graph g in which the proofs for each node are uniquely determined (i.e., $|Proofs(\mathtt{m}[B])| = 1$) and all method variants of g are still represented (i.e., the proof plan has all sink nodes of g), which preserves the correctness of g.

Definition 7 (Proof Plan). *Let $g = (N, E)$ be a proof graph and $N_p \subseteq N, E_p \subseteq E$. A subgraph $p = (N_p, E_p)$ of g is a* proof plan *derived from g (denoted by $p \sqsubseteq g$) when each node in N_p has at most one incoming edge in E_p and $Sinks(p) = Sinks(g)$.*

In Figure 3, we show an example proof plan (marked dashed). In this plan, we chose to conduct only a small number of proofs. Alternatively, we could have chosen *all* nodes (corresponding to a collection of spanning trees) or *only* sink nodes (i.e., the minimal proof plan). In section 4, we discuss how the chosen proof plan may influence the verification effort. In contrast to an entire proof graph, a proof plan represents a single proof per method variant, as each node by definition has no more than one incoming edge. Once a proof plan p is selected, we can verify whether p is correct according to Definition 6 by conducting its complete proofs (i.e., proofs for sink nodes) and checking whether they are all closed.

As a theorem, we observe that all proof plans are correctness-equivalent; that is, they are all either correct or incorrect.

Theorem 1 (Equivalence of Proof Plans). *Let g be any proof graph and $p_1, p_2 \sqsubseteq g$ any proof plans. Then, $p_1^{\checkmark} \Leftrightarrow p_2^{\checkmark}$.*

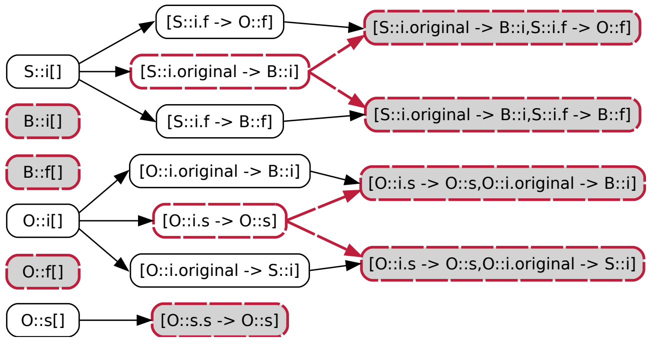


Figure 3: Simplified proof graph for the SPL from Figure 1.

Proof. By definition, all proof plans have the same sink nodes (those of g). Although the paths towards these nodes may potentially differ, Lemma 1 shows that differing paths (i.e., orders of adding bindings) lead to equivalent proofs. □

To relate proof graphs and plans to the correctness of SPLs, we describe how to construct the SPL proof graph. An SPL proof graph represents all method variants of an SPL and corresponds to many different strategies to verify an SPL (cf. section 4). In an SPL proof graph, each source node $\mathtt{m}[\varnothing]$ corresponds to a method $\mathtt{m}$ in the given SPL and spans one connected component. Each binding of a call $\mathtt{m.n}$ to a method $\mathtt{m}'$in the SPL then corresponds to some edges in this proof graph: That is, we draw an edge from some node to another node if the former node *admits* a binding of $\mathtt{m.n}$ and the latter binds $\mathtt{m.n}$ to $\mathtt{m}'$. We say that a node admits a binding when the bound call occurs in the connected component's method (i.e., the binding is syntactically meaningful) and when the resulting combination of bindings occurs in at least one configuration. By this construction, each source node (i.e., each method) branches into multiple paths towards some sink nodes (i.e., method variants).

The formal definition of SPL proof graphs is then as follows:

Definition 8 (SPL Proof Graph). *Let $\ell = (F, C, M)$ be an SPL. The* SPL proof graph *for ℓ is given by*

$$g(\ell){:=}(N_{B(\ell)}, \{(\mathtt{m}[B], \mathtt{m}[B']) \in N_{B(\ell)} \times N_{B(\ell)} \mid B \subset B'\}),$$

$$\text{where } N_{\varnothing}{:=}\{\mathtt{m}[\varnothing] \mid \mathtt{m} \in M, \exists c \in C\colon \mathtt{m} \in M(c)\},$$

$$N_{\hat{B} \cup \{b\}}{:=}N_{\hat{B}} \cup \{\mathtt{m}[B \cup \{b\}] \mid \mathtt{m}[B] \in N_{\hat{B}}, \mathtt{m}[B] \Vdash b\},$$

and a node $\mathtt{m}[B]$ admits *the binding $b = \mathtt{m}'.\mathtt{n} \mapsto \mathtt{m}''$ (denoted by $\mathtt{m}[B] \Vdash b$) iff $\mathtt{m}'.\mathtt{n} \in Calls(\mathtt{m}) \land \exists c \in C\colon (\mathtt{m} \in M(c) \land B \cup \{b\} \subseteq B(c))$.*

Definition 8 induces an algorithm for calculating the entire SPL proof graph. In Figure 3, we show this proof graph for the example SPL in Figure 1 (for better readability, we omit transitive edges). The source nodes in this graph are added by $N_{\varnothing}$ for each method in the SPL (e.g., B::i[], which abbreviates *Base*::ins[$\varnothing$]). Then, we successively construct the set $N_{\hat{B} \cup \{b\}}$ for every binding b in any configuration of the SPL (i.e., bindings in $B(\ell)$). That is, for each binding, we add nodes for all existing nodes that admit the binding ($\Vdash$). Finally, we connect all nodes according to the subset relation on their bindings. In the resulting graph in Figure 3, each sink node (marked gray) corresponds to one variant of a method in the SPL. For example, the method *Base*::ins has only one variant, which is included in several configurations of the SPL and represented by the node $Base::\mathtt{ins}[\varnothing]$. $Ord::\mathtt{ins}$, on the other hand, has two variants (for insertion into lists and sets, respectively), therefore there are two sink nodes in the subgraph starting at $Ord::\mathtt{ins}[\varnothing]$.

To leverage proof reuse for methods with multiple variants (e.g., $Ord::\mathtt{ins}$), we inspect the intermediate nodes in between source and sink nodes. For example, both sink nodes for $Ord::\mathtt{ins}$ are preceded by the node $Ord::\mathtt{ins}[\{\, Ord::\mathtt{ins.sort} \mapsto Ord::\mathtt{sort}\}]$. Thus, they share the verification effort invested in said node, so we may first conduct a proof for said node and then reuse that proof twice to arrive at both sink nodes. By choosing some nodes and edges for the other methods in the SPL, we get a concrete proof plan (e.g., the one marked dashed in Figure 3). Thus, we can verify the example SPL by conducting 10 proofs, two of which allow for partial proof reuse. The SPL is then correct if all proofs conducted for sink nodes (each corresponding to a method variant) are closed.

In the following, we show formally that the correctness of proof plans (cf. Definition 7) indeed coincides with the correctness of SPLs (cf. Definition 4) to justify the correctness of our approach.

Theorem 2 (Proof Plan Correctness). *Let $\ell = (F, C, M)$ be any SPL and $p \sqsubseteq g(\ell)$ any proof plan. Then, $p^{\checkmark} \Leftrightarrow \ell^{\checkmark}$. That is, an SPL is correct iff any proof plan derived from its proof graph is correct.*

Proof. All proof plans are correctness-equivalent (cf. Theorem 1), so it suffices to find a single proof plan p with $p^{\checkmark} \Leftrightarrow \ell^{\checkmark}$. Consider the minimal proof plan p_{op}, which consists of the sink nodes of $g(\ell)$ and no edges. By Definition 6 and Definition 8, each sink node $\mathtt{m}[B]$ in this plan corresponds to exactly one variant of the method $\mathtt{m} \in M$ in some configuration $c \in C$ such that $B \subseteq B(c)$.

Thus, it only remains to be shown that a sink node's proof $Proofs(\mathtt{m}[B]) = \{\bar{\Phi}\}$ is equivalent to the proof for its method variant; that is, $\bar{\Phi} = \{P \rightarrow \langle S \rangle\, Q\} \xrightarrow{B} \varnothing$ iff $\{P \rightarrow \langle S \rangle\, Q\} \xrightarrow{B(c)} \varnothing$. This follows because $B \subseteq B(c)$ and all bindings in $B(c) \setminus B$ do not affect any proof for $\mathtt{m}$ (cf. Definition 1). □

Theorem 2 justifies the *correctness* of our approach. However, it makes no statement about the *verification effort* associated with a proof plan. In fact, proof plans may largely differ in their verification effort. Consider, for example, the extreme examples mentioned above: If we chose to conduct proofs for *all* nodes in the SPL proof graph, we would conduct several proofs that do not correspond or contribute to the proof of any method variant, which would have a negative impact on the verification effort. On the other hand, choosing to conduct proofs *only* for sink nodes leaves room for improvement as well, as no proof reuse is leveraged.

Our notion of proof plans lays the foundation for finding a middle ground: Ideally, we would like to maximize proof reuse, while minimizing the number of partial proofs and thus, their overhead [21].

4 SPL VERIFICATION STRATEGIES

The flexible definition of proof plans allows us to implement several concrete verification strategies [33], which we discuss in the following in the context of proof plans.

Definition 9 (SPL Verification Strategies). *Let ℓ be an SPL and $g = g(\ell)$ its SPL proof graph. We introduce three particular kinds of proof plans derived from g:*

- $p_{op}{:=}(Sinks(g), \varnothing)$ *is the* optimized product-based *proof plan,*
- $p_{fp}{:=}(Sources(g) \cup Sinks(g), \{(\mathtt{m}[\varnothing], \mathtt{m}[B]) \mid \mathtt{m}[B] \in Sinks(g), B \neq \varnothing\})$ *is the* feature-product-based *proof plan, and*
- $p_{ff} = (N, E) \sqsubseteq g$ *is a* feature-family-based *proof plan when it is not (feature-)product-based and each non-sink node in N has at least two outgoing edges in E.*

A product-based strategy verifies each method variant in isolation, thus it does not leverage proof reuse. As noted in the proof for Theorem 2, there is a minimal proof plan p_{op} for any SPL, which implements the product-based verification strategy given in Definition 4. In this proof plan, we conduct one proof for each sink node (i.e., method variant). Because we ignore identical occurrences of a variant in several configurations, our strategy can be considered optimized, in contrast to one that reproves such occurrences [33].

Product-based strategies often serve as a baseline for evaluating SPL analysis strategies, as they are simple to implement, sound, and complete; however, they do not scale to large SPLs [33]. Instead, one step towards increased proof reuse (and, thus, possibly a reduction in verification effort) is to conduct partial proofs without any bindings for each method in an SPL, which we then complete with the bindings for each individual method variant. The partial proofs may then reduce the proof obligation set for a method such that only proof obligations related to (yet unbound) calls remain. In terms of proof plans, this corresponds to taking the source and sink nodes of a proof graph and drawing edges between them (p_{fp}). This proof plan implements a feature-product-based strategy [33], as it consists of a feature-based phase (verifying methods in isolation) followed by a product-based phase (verifying actual method variants based on the results from the feature-based phase).

With a proof plan such as the one shown in Figure 3 (marked dashed), we can continue the idea of p_{fp} to increase proof reuse even further by relying on intermediate (i.e., non-source and non-sink) nodes. That is, all non-sink nodes included in such an improved proof plan (p_{ff}) must have at least two outgoing edges, which means that each conducted partial proof is reused to verify at least two distinct method variants (as seen in Figure 3). We argue that this way of using intermediate nodes to construct partial proofs can be considered a family-based phase, making this a feature-family-based strategy [33]. This is because we no longer consider individual method variants in isolation as above; instead all conducted partial proofs contribute to the verification of several method variants.

In summary, we can understand these verification strategies—product-, feature-product-, and (arguably) feature-family-based—as special cases of our technique for SPL verification. As there are many proof plans for a given SPL, this shows that there is a nuanced spectrum of SPL verification strategies in between these known strategies and we can construct hybrids from the proof plans given in Definition 9: For example, such a hybrid plan may be product-based in one connected component of the proof plan and feature-family-based elsewhere. In practice, this means that we can combine and leverage the individual strengths of the discussed strategies. On a theoretical level, we show that there are nuances to SPL verification, which have (to the best of our knowledge) not been considered before. This also allows us to judge the degree to which a verification strategy (given as a proof plan) is product-, feature-product-, or feature-family-based.

$$\begin{tikzcd}
\underset{\text{Def. 2–4}}{\ell} \arrow[r, "\text{Def. 8}"'] \arrow[rr, dashed, bend left] \arrow[d, dotted, no head, "\simeq"] & \underset{\text{Def. 5–6}}{g(\ell)} \arrow[r, "\text{Def. 9}"'] \arrow[d, dotted, no head, "\simeq"] & \underset{\text{Def. 7}}{p} \arrow[d, dotted, no head, "\simeq"] \\
\ell' \arrow[r, dotted] \arrow[rr, dashed, bend right] & g(\ell') \arrow[r, dotted] & p'
\end{tikzcd}$$

Figure 4: Using proof plans for post-hoc verification (solid), evolution scenarios (dotted), and lazy exploration (dashed).

Besides the discussed strategies, there are two widely known SPL analysis strategies that are not representable as proof plans, namely feature-based verification [6] and family-based verification with metaproducts [35]: First, feature-based strategies verify each feature (in our context, each method) in isolation, ignoring all references to other features (here, all calls). In terms of proof graphs, this would mean conducting proofs only for the proof graph's source nodes, not for intermediate or sink nodes. However, this contradicts that all sink nodes must be included in a proof plan, which reflects the incompleteness of feature-based strategies (i.e., they can detect some, but not all errors) [33]. Second, family-based strategies encode all method variants into a single metaproduct, which is then verified as a whole. This strategy is incompatible with our approach because proof plans and metaproducts follow opposing ideas: While proof plans split the verification into many small, separate proofs, metaproducts only require a single, monolithic proof. Thus, proof plans allow for parallelization and evolution (see below); while proofs for metaproducts can profit from internal optimization.

5 APPLICATIONS

In Figure 4, we show three potentially useful directions for applying proof plans in practice, which we describe in more detail.

- *Post-Hoc Verification.* In this paper, we are mostly concerned with post-hoc verification (solid arrows in Figure 4). That is, we have an SPL ℓ, and to check it for correctness, we calculate its proof graph $g(\ell)$ and derive some proof plan $p \sqsubseteq g(\ell)$ (cf. Definition 2–9). In this scenario, we profit from proof reuse due to variability in space [2], as similar products share proofs.
- *Evolution Scenarios.* In practice, variability in time [2] is usually also involved (dotted arrows in Figure 4). This includes scenarios like evolution, maintenance, debugging, focusing on critical products, safe feature removal, and correctness-by-construction [23]. In such scenarios, ℓ is known to be (in-)correct due to some proof plan p and we want to determine whether an evolved SPL ℓ' is still (in-)correct. Assuming that ℓ' is quite similar to the original ℓ ($\ell \simeq \ell'$), their proof graphs can be expected to be similar as well ($g(\ell) \simeq g(\ell')$). Thus, a proof plan p' similar to p ($p \simeq p'$); should be derivable, and we also profit from proof reuse due to variability in time. Notably, family-based verification does not support evolution scenarios well, in contrast to proof plans.
- *Lazy Exploration.* Calculating the entire proof graph $g(\ell)$ according to Definition 8 is infeasible for larger SPLs. Thus, we must consider skipping its calculation and instead explore the space of possible proof plans lazily. To this end, we need an algorithm to derive a proof plan p directly from its SPL ℓ (dashed arrows in Figure 4). However, as we now skip manually choosing a proof plan, heuristics for determining a "good" proof plan (or optimizing an existing one) are needed as well. Depending on the use case, a "good" proof plan may be derived by strictly maximizing branching (cf. Definition 9) or predicting how possible future versions ℓ' might evolve to minimize costly re-verification.

Because a complete description and evaluation of the second and third application would require further definitions (e.g., similarity), theorems (e.g., $g(\ell) \simeq g(\ell')$), and algorithms (e.g., for constructing p'), we leave those to be investigated in future work.

6 EVALUATION

To demonstrate feasibility, we implemented proof plans in the prototype *KeYPl* (*KeY for Proof Plans*).[2] We use the KeY verification system [1] as underlying verification calculus and the Java modeling language (JML) [25] for specification. As KeY has no built-in support for partial proofs, we implement them with *model methods*, which are methods on the specification-level [1]. In particular, model methods can be abstract, which allows us to pause a proof until a concrete definition is substituted. *KeYPl* completely automates the process of generating JML-annotated Java code and performs the necessary substitutions to implement partial proofs. Our prototype implements a slightly expanded version of our formal concept that also supports composition of method contracts via explicit contract refinement [24, 34].

We use our prototype to evaluate the verification effort required by proof plans in a post-hoc verification scenario (solid arrows in Figure 4). That is, we aim to create proof plans to verify a given SPL using all verification strategies described in Definition 9, after which we compare their verification effort.

Experimental Setup. As a subject system, we developed a case study of list data structures, which is an extension of our example SPL in Figure 1. The case study comprises 5 features and 13 methods with 16 calls in total (11 of which are `original` calls, 7 of those in contracts). With *KeYPl*, we can calculate its entire SPL proof graph according to Definition 8, which consists of 37 nodes.

We use *KeYPl* to derive proof plans from this graph for the verification strategies discussed in section 4 as follows: First, the optimized and feature-product-based proof plans p_{op} and p_{fp} are both uniquely determined (cf. Definition 9). Second, we calculate *all* feature-family-based proof plans p_{ff} (to ensure reproducibility)—as the proof graph is small, we do this by iterating all proof plans (cf. Definition 7) and filtering those that are feature-family-based (cf. Definition 9), yielding 14 in total. Third, we emulate an unoptimized product-based strategy, which comprises the same proofs as p_{op}, but does not eliminate strict duplicates [33]. Finally, to compare *KeYPl* with a family-based strategy, we also generate the metaproduct [35] for our case study with FeatureHouse [4]. We then verify all derived proof plans (cf. Definition 6) and the metaproduct with KeY and trace the number of open (i.e., failed) proofs. We further measure the required verification effort with commonly used metrics, namely the number of KeY-internal proof steps/branches and the required time [1, 22]. We perform our evaluation with 5 repetitions (plus JVM warm up) on a quad-core 2.3 GHz CPU with 12 GB of RAM and were able to reproduce the results on another machine.

[2]Prototype, case study, and results available at: https://github.com/ekuiter/KeYPl

Table 1: Comparison of verification strategies for our case study (w.r.t. open proofs, KeY-internal steps/branches, time).

Verification strategy	#Open	#Step	#Branch	Time [s]
Metaproduct [35]	2	18350	245	20.0
p_{op}	0	21400	281	23.7
p_{ff} (avg., $N = 14$)	0	24217	313	26.2
p_{fp}	0	25921	348	28.8
Unoptimized	0	148528	1964	157.7

Results and Discussion. In Table 1, we show the results of our evaluation, ordered by increasing verification effort. For the feature-family-based strategy, we show the arithmetic mean of all 14 derived proof plans (with a negligible standard deviation of 201 proof steps and 0.8s). We observe that the family-based strategy, which is not based on partial proofs, performs best in terms of verification effort (20s). In particular, this strategy requires less verification effort than all partial-proof-based strategies, which is in line with previous evaluations of partial proofs [8, 21]. However, the proofs for two (of 13) methods (i.e.,`insert` and `remove`) cannot be closed by this strategy, although the metaproduct was correctly generated (which we checked manually). We suspect that this is due to the complexity of those methods, which may push KeY to its limits.

The following three strategies are based on proof plans and all have comparable verification effort (24-29s). Surprisingly, the feature-family-based approach with partial proofs for intermediate nodes performs worse than the optimized product-based approach, which suggests that partial proof reuse could not be attained as intended. Finally, the unoptimized product-based strategy performs considerably worse than all other strategies, as expected (158s).

In summary, our evaluation shows that proof plans perform worse than metaproducts in a post-hoc verification scenario; however, they are able to compete at least in the same magnitude.

Threats to Validity. Regarding internal validity [37], due to the two open proofs for the family-based strategy, we only have a lower bound on the measured time (20s), which may in truth be higher.

Regarding construct validity, our results show that partial proof reuse could not be successfully achieved in the feature-family-based proof plans. Thus, we encountered similar issues as others in implementing partial proofs in KeY [15, 21, 28]. In particular, the optimized product-based strategy outperforms all strategies with partial proofs, suggesting that the overhead imposed by our implementation of partial proofs outweighs the potential for proof reuse. However, this issue does not threaten the validity of our concept, as our evaluation only concerns one implementation of partial proofs (i.e., model methods) in one verification system (i.e., KeY).

Regarding external validity, first, we focus on a single case study, which limits the generalizability of our findings. However, this is acceptable as case studies for deductive verification of SPLs are rare and difficult to construct from scratch: We are only aware of two such case studies (`IntList` and `BankAccount`), both of which are smaller (in terms of specified methods) and less complex (in terms of `original` calls) than our case study. Second, we only evaluate the post-hoc verification scenario from section 5, as only this application is thoroughly covered by Definition 2–9. In evolution scenarios, proof plans can be expected to perform much better than metaproducts due to the added variability-in-time proof reuse.

7 RELATED WORK

Bubel et al. [8] propose proof repositories as a framework for proof reuse. Similar to our work, they aim to reduce verification effort for many similar method implementations by conducting and completing partial proofs. Their framework is generalizable to several notions of compositionality. Our work is inspired by proof repositories, but we significantly extend the idea of Bubel et al.: First, we adapt their general framework to a concrete use case (i.e., feature-oriented SPLs), prove its correctness, and evaluate it on a realistic instance. Second, proof repositories are nondeterministic; whereas we clearly distinguish proof graphs and plans and discuss the resulting spectrum of verification strategies. Third, proof repositories are conceptually coupled to abstract contracts, an implementation of partial proofs. In contrast, we provide the first systematic formalization of partial proofs, bindings, and proof reuse.

Knüppel et al. [21] introduce a partial-proof-based technique for proof reuse in evolving FOP-based software systems. Their algorithm Fefalution is feature-family-based and completes partial proofs using a metaproduct. Compared to this work, they strictly limit their usage of partial proofs to the feature-based phase, while we also consider a potentially large number of intermediate partial proofs. Thus, we focus mostly on proof reuse by partial proofs, while they aim for a symbiosis of partial proofs and metaproducts.

Klebanov [19] and Thüm et al. [36] propose techniques for proof reuse (proof replay and proof composition, respectively) that operate on a proof-step level. Their approaches are more flexible than ours, but require hand-written proofs. Hähnle et al. [17] introduce abstract contracts as an implementation of partial proofs in a verification system. They extend the verification calculus with a new rule that allows for separating a call from the called method. Steinhöfel and Hähnle [32] propose a generalization of abstract contracts, abstract execution, for proving properties about an infinite number of programs. Our technique works on a meta-level: We describe how to utilize partial proofs once implemented.

8 CONCLUSION

In this paper, we proposed proof plans as a partial-proof-based technique for verifying feature-oriented SPLs. We showed how proof graphs span a space of all possible proofs, how proof plans represent a spectrum of SPL verification strategies, and how to apply proof plans in practice. We briefly described a prototype that implements proof plans using the KeY verification system and evaluated the verification effort of proof plans on a case study. Our work is the first to fully automate partial proofs; thus, it constitutes an important step towards aligning software reuse with proof reuse.

In future work, we aim to improve our implementation of partial proofs in KeY or another verification system. Thus, we want to decrease the overhead of partial proofs and achieve proof reuse in feature-family-based proof plans as well. Then, we may also implement advanced applications, such as discerning "good" proof plans with heuristics that predict and maximize the amount of proof reuse, efficient lazy exploration of the SPL proof graph, and evolution scenarios to synergize proof reuse over time and space.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification – The KeY Book*. Springer.

[2] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/3382025.3414955

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[4] Sven Apel, Christian Kästner, and Christian Lengauer. 2009. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering*. IEEE. https://doi.org/10.1109/icse.2009.5070523

[5] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 372–375. https://doi.org/10.1109/ase.2011.6100075

[6] 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 Feature Interaction in Communications and Software Systems.

[7] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23

[8] 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*, Bernhard Steffen (Ed.). Springer, 130–156. https://doi.org/10.1007/978-3-319-46508-1_8

[9] Richard Bubel, Reiner Hähnle, and Maria Pelevina. 2014. Fully Abstract Operation Contracts. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 120–134. https://doi.org/10.1007/978-3-662-45231-8_9

[10] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

[12] Ramez Elmasri and Shamkant Navathe. 2010. *Fundamentals of Database Systems*. Addison-Wesley.

[13] Shannon Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 391–400. https://doi.org/10.1109/ICSME.2014.61

[14] David Harel. 1979. *First-Order Dynamic Logic*. Springer. https://doi.org/10.1007/3-540-09237-4

[15] Marlen Herter-Bernier. 2021. *Verifikation Evolvierender Softwareproduktlinien mittels Uninterpretierter Prädikate*. Master's thesis. Technische Universität Braunschweig.

[16] 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

[17] Reiner Hähnle, Ina Schaefer, and Richard Bubel. 2013. Reuse in Software Verification by Abstract Method Calls. In *Proceedings of the International Conference on Automated Deduction*. Springer, 300–314. https://doi.org/10.1007/978-3-642-38574-2_21

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

[19] Vladimir Klebanov. 2007. Proof Reuse. In *Verification of Object-Oriented Software. The KeY Approach*, Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt (Eds.). Springer, 507–529. https://doi.org/10.1007/978-3-540-69061-0_13

[20] Peter Knauber, Jesús Bermejo Muñoz, Günter Böckle, Julio Cesar Sampaio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David M. Weiss. 2001. Quantifying Product Line Benefits. In *Proceedings of the International Workshop on Software Product-Family Engineering*. Springer, 155–163. https://doi.org/10.1007/3-540-47833-7_15

[21] 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*. Springer, 122–148. https://doi.org/10.1007/978-3-030-64354-6_5

[22] Alexander Knüppel, Thomas Thüm, Carsten Padylla, and Ina Schaefer. 2018. Scalability of Deductive Verification Depends on Method Call Treatment. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 159–175. https://doi.org/10.1007/978-3-030-03427-6_15

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

[24] Elias Kuiter. 2020. *Proof Repositories for Correct-by-Construction Software Product Lines*. Master's thesis. University of Magdeburg.

[25] Gary T. Leavens and Yoonsik Cheon. 2006. *Design by Contract with JML*. Technical Report. University of Texas at El Paso.

[26] Jing Liu, Josh Dehlinger, and Robyn Lutz. 2007. Safety Analysis of Software Product Lines using State-Based Modeling. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 10–30. https://doi.org/10.1109/issre.2005.36

[27] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[28] Maria Pelevina. 2014. *Realization and Extension of Abstract Operation Contracts for Program Logic*. Bachelor's thesis. Technische Universität Darmstadt.

[29] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[30] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming*. Springer, 419–443.

[31] Yannis Smaragdakis and Don Batory. 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology* 11, 2 (2002), 215–255. https://doi.org/10.1145/505145.505148

[32] Dominic Steinhöfel and Reiner Hähnle. 2019. Abstract Execution. In *Proceedings of the International Symposium on Formal Methods*. Springer, 319–336. https://doi.org/10.1007/978-3-030-30942-8_20

[33] 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

[34] 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

[35] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*. ACM, 11–20. https://doi.org/10.1145/2371401.2371404

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

[37] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer.