Delta-based Verification of Software Product Families

Marco Scaletta
Technical University of Darmstadt
Darmstadt, Germany
scaletta@cs.tu-darmstadt.de

Reiner Hähnle
Technical University of Darmstadt
Darmstadt, Germany
haehnle@cs.tu-darmstadt.de

Dominic Steinhöfel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
dominic.steinhoefel@cispa.de

Richard Bubel
Technical University of Darmstadt
Darmstadt, Germany
bubel@cs.tu-darmstadt.de

Abstract

The quest for feature- and family-oriented deductive verification of software product lines resulted in several proposals. In this paper we look at delta-oriented modeling of product lines and combine two new ideas: first, we extend Hähnle & Schaefer's delta-oriented version of Liskov's substitution principle for behavioral subtyping to work also for overridden behavior in benign cases. For this to succeed, programs need to be in a certain normal form. The required normal form turns out to be achievable in many cases by a set of program transformations, whose correctness is ensured by the recent technique of abstract execution. This is a generalization of symbolic execution that permits reasoning about abstract code elements. It is needed, because code deltas contain partially unknown code contexts in terms of "original" calls. Second, we devise a modular verification procedure for deltas based on abstract execution, representing deltas as abstract programs calling into unknown contexts. The result is a "delta-based" verification approach, where each modification of a method in a code delta is verified in isolation, but which overcomes the strict limitations of behavioral subtyping and works for many practical programs. The latter claim is substantiated with case studies and benchmarks.

CCS Concepts: • **Software and its engineering → Software product lines**; **Software verification**.

Keywords: Software product lines, delta-oriented programming, abstract execution, deductive verification, program transformation, behavioral subtyping

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487200

ACM Reference Format:
Marco Scaletta, Reiner Hähnle, Dominic Steinhöfel, and Richard Bubel. 2021. Delta-based Verification of Software Product Families. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486609.3487200

1 Introduction

In deductive software verification [20] programs are formally specified by contracts [33] at the granularity of methods or procedures. Contracts are written in a specification language based on first-order logic. Proof obligations are discharged with one of several available verification tools. The quest to lift deductive verification to feature- and family-oriented [39] approaches that work for software product lines (SPLs) resulted in various proposals, for example, [21, 22, 27, 41–43].

The fundamental design space of SPL verification is demarcated by two extremes: in the ideal scenario for *feature-oriented* verification one verifies base code and family-specific code separately for each feature. A suitable composition mechanism then guarantees correctness of each valid variant. The main drawback is that compositionality requires serious constraints on the admissibility of contracts and feature implementations. For example, Hähnle & Schaefer [21] proposed an adaptation of *Liskov's Substitution Principle* (LSP) [31] in the context of delta-oriented programming [35]. Further contract composition principles are discussed by Thüm et al. [41]. The problem with constraints on contract admissibility is that this can impose too severe restrictions on software design to be of practical use. On the other end of the design space is *product-based* verification, where each valid product is specified and verified in isolation. This is usually prohibitive in cost, particularly, with respect to specification [7]. Besides, it excludes systematic reuse, the purported main advantage of software product lines.

The approaches cited above strike differing trade-offs, but the fundamental insight is clear: the more localized, i.e., feature-oriented, a verification approach is designed to be, the more constraining assumptions on contracts and implementations are needed. In the present paper we import two

new ideas into the design of a fully compositional verification approach that is nevertheless practical.

Our frame of reference is delta-oriented programming (DOP) [35], a feature-oriented, generative implementation principle compatible with any OO programming language. In DOP each feature is implemented by one or more delta modules (deltas, for short) that are applied successively to a core variant. The choice of DOP derives from the fact that deltas specify incremental code transformations at the granularity of a method declaration. This matches contract-based specification. Hence, each modification of a method of a delta is assumed to be specified with a contract (see Listing 8 for an example). As usual in DOP, a delta for a method can be declared relative to a previous version of that method and called using the keyword **original** in the delta's code.

Like [21] and some of the approaches in [41] we impose restrictions on contracts and deltas to render their verification compositional. The new idea we propose is to combine these restrictions with a *normal form* that is enforced for the code declared in a delta. This normal form makes it possible to define more liberal constraints for deltas and contracts that *allow overriding* of behavior and are not compositional in the general case. We prove that the correctness of each local contract of a core method declaration and of all its modifications declared in a delta implies correctness of all valid variants that result from delta application.

The obvious drawback of imposing a normal form is that few implementations follow it. To mitigate this problem we define a set of *behavior-preserving* program transformations that can be used to achieve normal form. In our case studies the vast majority of cases required only a small number of transformation steps to the desired normal form. We also discuss one instance, where some more (but also limited) remodeling was required.

To verify correctness, two verification tasks remain: (1) For the employed program transformation schemata one needs to show that they preserve behavior; (2) all core method declarations and modifications in deltas must be proven correct relative to their contract. Both tasks involve verification of code that contains *abstract* segments: transformation schemata must be valid for all possible instances, while method modifications in deltas contain calls to an unknown **original** method. To handle abstract code we use a recent verification approach called *abstract execution* [38], a generalization of symbolic execution that permits reasoning about abstract code elements. It is implemented [36] in the deductive verification tool KeY [2]. With abstract execution we managed to verify all proof obligations resulting from transformation schemata and local contracts from our case studies in a fully automated manner. The transformation schemata need to be verified only once for all instances. From the validity of the local contracts, the correctness of all valid variants follows by our composition theorem.

As our approach is centered around method modifications declared in deltas, it cannot be classified purely as a feature- or family-based verification approach [39]: Unlike the former, a feature might be implemented in several deltas; unlike the latter, it suffices to verify each delta in isolation once a simple family-based analysis has checked admissibility. For this reason we prefer the terminology *delta-based* verification.

The paper is structured as follows: Section 2 provides background on specification and verification of abstract programs with abstract execution. This involves a small extension of the Java Modeling Language, which is another (minor) contribution of our paper. Section 3 gives a compact introduction to DOP to make the paper self-contained. We also explain how **original** calls are modeled with abstract execution. Section 4 rehashes the main results of the delta-oriented adaption of LSP [21]. This constitutes the baseline for the definition of the normal form and a liberalized composition principle in Section 5, where also the composition theorem is stated and proven. Section 6 presents the program transformation schemata required to achieve normal form. Section 7 evaluates our approach: first its principled feasibility, second its coverage, third its effectiveness in comparison with a product-based approach. Section 8 discusses related work. In Section 9 we conclude and mention future directions.

2 Specifying Abstract Programs

2.1 Abstract Execution

Abstract Execution (AE) [38] is a generalization of symbolic execution [10, 26] that makes it possible to prove properties of *abstract* programs containing *placeholder symbols*, representing potentially infinitely many concrete statements or expressions. For example, we can prove that after symbolically executing the abstract program "`P low = 17;`", the final value of variable `low` will be 17—for *any* initial value of `low` and for *any* concrete statement `P`, as long as `P` completes normally (does not throw an exception, etc.).

AE trades off automation with expressiveness. The semantics of Abstract Statements (AStmts) and Abstract Expressions (AExps) is represented by *abstract state changes* using dedicated symbolic execution rules. Instead of performing syntactic case distinctions as done by *structural induction* [10], the traditional approach to prove universal properties of abstract programs, AE only records the *behavior* of placeholder symbols. This restriction to *behavioral properties*, in connection with symbolic execution, frequently permits *fully automatic proofs*.

AE goes beyond the capabilities of existing approaches for automatically verifying abstract programs [9, 17, 32, 37]. These are confined to a specific application scenario and limited regarding expressiveness. Notably, AE supports AStmts *as well as* AExps and can reason about programs containing loops. Moreover, AE allows us to impose constraints on the behavior of the concrete statements or expressions

represented by placeholder symbols: one can specify (i) the locations that can be written by a placeholder (its *frame*), (ii) the locations that can be read by a placeholder (its *footprint*), (iii) the condition under which instances may complete abruptly (throwing an exception, breaking from a loop, returning, etc.), and (iv) state postconditions that must hold after executing an instance of a placeholder.

2.2 Abstract Programs, Statements, and Expressions

For simplicity we restrict ourselves to normally completing statements and expressions in this paper, and, accordingly, omit to specify explicitly that abrupt completion is excluded. Furthermore, we use a simplified syntax to specify frames and footprints of placeholder symbols. For example, to specify an AStmt with identifier symbol "`Stmt`" that may assign *at most* to variables `x` and `y`, while being allowed to access *at most* the initial values of variables `y` and `z`, we write "`Stmt(x, y :≈ y, z);`". For an abstract *expression* with identifier symbol "`expr`" of type T and the same frame and footprint as above we write "$\texttt{expr}^T$`(x, y :≈ y, z)`". By convention, we use upper case identifiers for AStmts, and lower case for AExps. Both the frame (left of ":≈") and footprint (right of ":≈") in such declarations may be empty.

Consider again the example "`P low = 17;`". It is generally *incorrect* that the final value of `low` will be 17 if we swap the order of the concrete and abstract statement, since there are instantiations of `P` assigning a value different from 17 to `low`. But we *may* swap these statements, if we restrict `P` such that its instances do not assign `low`. The machinery introduced so far is insufficient to accomplish this in a general manner: we are forced to specify an AStmt with an *empty* frame to ensure that `low` is not overwritten, and with footprint `\everything`, because we do not know the actual variables or fields that occur in the frames and footprints of the instances of `P`. The first is overly restrictive, the second overly defensive. To overcome this, we leverage the *theory of dynamic frames* [25]. A dynamic frame is an abstract specification variable that evaluates to a *set* of concrete locations. Let *frameP* and *footprintP* be dynamic frames. Then, the value of `low` after executing "`low = 17; P(`*frameP* :≈ *footprintP*`);`" will be 17 if we add the assumption `low` $\notin$ *frameP*. The semantics of the AStmt `P(`*frameP* :≈ *footprintP*`)` is the set of all concrete statements which read arbitrary locations and do not assign `low`.

We formally define the semantics of abstract programs.

Definition 2.1. An *Abstract Program Element (APE)* is either an AStmt or an AExp. A triple $\mathcal{P} = (locs, APEs, p_{abstr})$ of a set of dynamic frames *locs*, abstract program element declarations $APEs \neq \emptyset$ using these location sets, and a program fragment p_{abstr} containing exactly those APEs is called an *abstract program*. Each APE declaration is either an AStmt declaration `Stmt(`*frame* :≈ *footprint*`)` or an AExp declaration $\texttt{expr}^T$`(`*frame* :≈ *footprint*`)`, where `Stmt` and `expr` are identifiers, T is a type, and *frame*, *footprint* are location lists (program variables, fields, or dynamic frames).

A concrete program fragment p is a *legal instance* of $\mathcal{P}$ if it arises from substituting (in p_{abstr}) concrete sets of locations for all dynamic frames in *locs* and concrete statements or expressions for all APEs in *APEs*, where (1) all APEs are instantiated by concrete program elements respecting their frame, footprint, and (in the case of APEs) type; (2) all APEs with the same identifier are instantiated with the same concrete statement or expression. The semantics $[\![\mathcal{P}]\!]$ consists of all legal instances.

Restricting instances of APEs of the same identifier to the same concrete program elements is crucial for using AE to specify and prove *transformation rules*. For example, we can prove that the semantics of the abstract program "`P(`*frame* :≈ *footprint*`); P(`*frame* :≈ *footprint*`);`" is the same as that of "`P(`*frame* :≈ *footprint*`);`", whenever it holds that *frame* $\cap$ *footprint* $= \emptyset$.

2.3 Specification Annotations

Methods can be equipped with preconditions, which are assumed in correctness proofs, and postconditions, which state the proof goal. We use the Java Modeling Language's (JML) [2, 30] specification comments starting with an "`@`" symbol. Consider method "`Unit update(Int x)`" for updating the balance of a bank account. To specify that only positive updates are considered, we can add the specification line "`//@ requires x > 0;`" before the method declaration. If our goal is to prove that the field "`balance`" has been updated accordingly after the call we write

```
//@ ensures balance = \old(balance) + x;
```

The "**`\old`**" keyword permits referring to the value locations had just before method execution. The values of abstract location sets for AE can be constrained in method preconditions, but also within method bodies using the keyword **`ae_constraint`** instead of **`requires`**. To specify that two location sets *frame* and *footprint* are disjoint, for instance, we write "`//@ ae_constraint` *frame* $\cap$ *footprint* $= \emptyset$;". For the sake of readability, we use conventional mathematical operators like "$\cap$" and not the JML equivalents like "**`\intersect`**".

The JML reference manual [30, Sect. 12.4.2] defines an extension "**`\old`**`(`*expr*`, `*label*`)`" of the "**`\old`**" specifier to refer to the value an expression *expr* had when the control flow last reached the code position at *label*. According to the reference manual, labeled **`\old`** expressions may only be used in assumptions and assertions *within* method bodies, and only when *label* has been declared in the surrounding context. We added support for this previously unimplemented JML feature in the deductive verifier KeY [1]. Going beyond the JML definition, we allow labeled **`\old`** as well in *method postconditions*, whenever the given label is uniquely defined within the annotated method's body.

2.4 Transformation Rules

Schematic code transformation rules can be naturally expressed as pairs of abstract programs. For instance, the *Slide Statements* refactoring technique [16], which swaps two consecutive statements, can be expressed as "`P Q` ⇝ `Q P`", where `P` and `Q` are abstract statements. Using AE, we can impose constraints on APEs ensuring that a transformation rule is semantics-preserving for all concrete instances of the abstract programs. In the case of *Slide Statements*, the semantics of "`x = y; y = x;`" is not preserved when swapping the two statements, because each statement writes to the location the other one reads. Likewise, applying the refactoring to "`x = y; x = z;`" should be forbidden, because each statement overwrites the other one's assignment. The correct instances of *Slide Statements* for normally completing programs are modeled with our AE notation as:

```
/*@ ae_constraint                          /*@ ae_constraint
  @   frA ∩ frB = ∅ &&                     @   frA ∩ frB = ∅ &&
  @   frA ∩ fpB = ∅ &&          ⇝          @   frA ∩ fpB = ∅ &&
  @   frB ∩ fpA = ∅; */                    @   frB ∩ fpA = ∅; */
A(frA :≈ fpA)                            B(frB :≈ fpB)
B(frB :≈ fpB)                            A(frA :≈ fpA)
```

The REFINITY workbench[1] is based on KeY [1] and permits to specify statement-level transformation rules using the AE framework. Apart from providing editing support, it generates suitable proof obligations for the KeY prover to ensure semantic equivalence of the source and target abstract programs. For details on REFINITY and the construction of proof obligations for correctness proofs of transformation rules, we refer to [36].

3 Delta-Oriented Programming

Delta-oriented programming (DOP) [35] is a feature-oriented, generative approach to the design of software product lines. There are two major implementations: For Java [28, 44] and for the active object language ABS [11]. Because the former is currently being updated to recent Java versions, we use ABS, however, it does not matter greatly, because ABS has a Java-ish syntax and we focus on its sequential Java fragment in this paper.

In DOP [35], a family of programs is represented by a *delta model D* consisting of a base variant (*core module*) *C* and a partially ordered set of *delta modules* (deltas) that can be applied to the *core* to generate different variants. For example, in the simple delta model of a product line of bank accounts in Listing 1 the core module is called `BankAccount` and consists of an equally named class. There is a single delta module called `DFee` that is supposed to implement transaction fees.

Variability is modeled with *features*, abstract program characteristics. Each product of an SPL is represented by a set

[1] https://www.key-project.org/REFINITY/

```
1  class BankAccount {
2    Int balance;
3    Unit update(Int x) { balance = balance + x; }
4  }
5  delta DFee {
6    modifies class BankAccount {
7      adds Int fee;
8      modifies Unit update(Int x) {
9        original(x);
10       if (x < 0) { balance = balance - fee;}
11     }
12   }
13 }
14 productline SimpleSPL {
15   features BankAccount, Fee;
16   delta DFee when Fee;
17 }
```

Listing 1. A small portion of SimpleSPL

of features, called *feature selection* or *configuration*. The *feature model* [24] of an SPL specifies which combinations of features, i.e., which products, are legal. This is often done as a combination of a tree-shaped feature diagram [4] and Boolean constraints, but the latter is sufficient (diagrams are only needed to enhance understanding). The SPL in Listing 1 has two features `BankAccount` and `Fee` (Line 15) with the trivial constraint that the former is mandatory and the latter is optional (the constraint is not given in the listing). Obviously, each of the products P_1 = {`BankAccount`}, P_2 = {`BankAccount`, `Fee`} is legal, but P_3 = {`Fee`} is not.

In DOP each feature is associated with one or more deltas that are supposed to implement it (for example, Line 16). A legal selection of a set of features (i.e., a product) triggers activation of an associated set of deltas $\delta_1, ..., \delta_p$ to be applied to the core module, written $C\delta_1 \cdots \delta_p$. For example, generation of product P_2 can be denoted "`BankAccount DFee`". The sequence of delta applications must respect the specified partial order among the deltas (trivial in the example, because there is just one delta). If no valid delta application order can be found, then the requested product is invalid.

A delta can alter the core or the result of a previous delta application by adding, removing, and modifying classes. The internal structure of a class can be changed by adding (Line 7) and removing fields and methods, as well as modifying (Lines 8–11) the implementation of existing methods.

By calling **original** (Line 9) in the body of a method it is possible to refer to the most recent version of the method that is being modified. It is important to note that this most recent version is not uniquely determined until a product variant and its delta application order is fixed. Therefore, in general the number of versions of a method that a call to **original** represents is exponential in the number of features. We refer to this phenomenon as the *variant explosion problem* below. As its consequence, **original** calls are the pivotal point that must be considered in family-based analysis of DOP [42].

Modeling Original Calls with Abstract Execution

A main idea behind our approach is a new method to model **original** calls: we represent them as a call to an *abstract* version of the modified method with an abstract frame and partially [2] abstract footprint, i.e., an abstract program in the sense of AE. With the AE machinery we can then specify partially abstract constraints. We model the abstract nature of a (void) **original** call with parameters $\overline{\mathtt{p}}$ as an AS `Original` with its `frame` and `footprint`. We use the syntactic sugar `original(`$\overline{\mathtt{p}}$`)` to represent `Original(frame` $:\approx$ `footprint)` with the constraint $\overline{\mathtt{p}} \in$ `footprint`.

4 Behavioral Subtyping

Method *contracts* [33] are a standard approach to specifying the behavior of OO programs [20] that decomposes a global specification into method-level annotations called *contract*:

Definition 4.1. A *contract* for a method `m` is a triple with the notational convention (`m.r`, `m.e`, `m.frame`), where (1) `r` is a first-order formula called *precondition* or requires clause; (2) `e` is a first-order formula called *postcondition* or ensures clause; (3) `frame` is a set of program locations in `m`, whose value can potentially be changed during execution of `m`.

4.1 Liskov's Substitution Principle for DOP

The variant explosion problem pointed out above is an obvious challenge to specifying with contracts and to family-based verification in DOP, because the contract of a method in an **original** call is known only when the concrete variant is generated. Hähnle & Schaefer [21] addressed it by adapting *behavioral subtyping* [31], namely that the properties of an object must hold for all its subtypes, to DOP. A sufficient condition to ensure behavioral subtyping is commonly known as *Liskov's Substitution Principle*. In the realm of DOP it can be conveniently expressed by ordering contracts according to generality [21]:

Definition 4.2. Let `m`, `m'` be methods with contracts (`m.r`, `m.e`, `m.frame`), (`m'.r'`, `m'.e'`, `m'.frame'`), respectively, (`m` = `m'` permitted). The first contract is *more general* than the second (the second is *more specific* than the first), denoted

$$(\mathtt{m.r},\ \mathtt{m.e},\ \mathtt{m.frame}) \geq (\mathtt{m'.r'},\ \mathtt{m'.e'},\ \mathtt{m'.frame'})$$

iff the following holds:

$$(\mathtt{m.r} \rightarrow \mathtt{m'.r'}) \wedge (\mathtt{m'.e'} \rightarrow \mathtt{m.e}) \wedge (\mathtt{m'.frame'} \subseteq \mathtt{m.frame})$$

Definition 4.3. Let δ`.m` denote the code changes applied by a delta δ to method `m`. We say that δ`.m` fulfills *Liskov's Substitution Principle* (LSP) if the contract of δ`.m` is more specific than the contract of each previous version of `m`. A delta fulfills LSP if each of its methods does.

[2] Because of method parameters: these can be viewed as concrete members of the footprint without side effects. If a method has no parameters the footprint of its **original** call is fully abstract.

4.2 Verification of Deltas with the LSP

To verify that a code delta δ`.m` fulfills its contract the method calls inside δ`.m` are analyzed as follows. For each called method `n` (if the call is via **original**, then `n` is a version of `m` outside δ) there are two options: `n` occurs in δ or it does not. In the first case, the contract of δ`.n` is used. Otherwise, we search the given partial delta order for the delta, where `n` was added the last time before the current call (a method can be removed and added back) and use the contract of that version. LSP ensures this contract is fulfilled by all possible versions of `n` until the current call, because subsequent contracts of `n` can only become more specific.

Definition 4.4. We say that a delta δ is *LSP-verified* if all methods in δ satisfy their contract, where the contract of called methods is chosen as specified above.

For a somewhat more rigorous definition, see [21], where the terminology *verified delta* instead of *LSP-verified delta* is used. We use the latter for clarity. Also in [21] it was proven that if the core of an SPL satisfies its contracts, each delta fulfills LSP and is LSP-verified, then all product variants satisfy their contracts. In what follows, we refer to this kind of modular verification as *Liskov-modular verification*.

Let N and M be the number of deltas and different methods, respectively. With Liskov-modular verification the number of contracts that need to be proven to verify an SPL is polynomial in N and M, $O(N * M)$, instead of $O(2^N * M)$, needed by a product-based analysis. This reduction of complexity is due to transitivity of the more-general-than order between contracts: given a delta δ modifying method `m`, we do not need to consider each previous version of `m`, but only the contract of the most general one in the sense of Definition 4.4.

5 Beyond LSP

As LSP is too strict, it is often (partially) violated by real-world programs. Any verification methodology relying on LSP will thus reject large classes of programs even though they are correct and well designed. Hence, it is of great interest to extend this range to a larger class of programs. We do this in three parts: (i) to ensure fully local specification and verification of deltas, we define a normal form for programs with **original** calls; (ii) we define a number of "benign" conditions that violate LSP, but adhere to natural patterns for the design of deltas, and are still sufficient to guarantee soundness of all variants; and (iii) we show that many delta models satisfying neither LSP nor our more liberal conditions can be naturally transformed into models satisfying our conditions by just a few code refactorings. We describe the two former steps in the current section and the latter in Section 6.

```
// refactorable to SSONF
modifies Unit m(Int x) {
  if (x > 0) {
    field1 += 1;
    m(); // BP-method
    original(x);
  } else {
    original(x);
    field2 += 1;
  }
}
```

```
// refactored in SSONF
modifies Unit m(Int x) {
  if (x > 0) {
    field1 += 1;
    m(); //BP-method
  }
  original(x);
  if (x<=0) {
    field2 += 1;
  }
}
```

Listing 2. A modifying delta & its refactored SSONF version

5.1 Normal Form

To prepare our results we introduce the concept of a *behavior preserving* method.

Definition 5.1. We say that a method `m` is *behavior preserving* (a BP-method) if each method it calls is itself behavior preserving and one of the following holds:

- All code modifications of `m` fulfill LSP (Definition 4.3),
- `m` is never modified.

We call a sequence of statements without calls to non BP-methods *behavior preserving sequence* (BP-sequence). Specifically, the body of a BP-method consists of a BP-sequence.

During verification BP-methods can be handled as described in Section 4, i.e., when a BP-method `m` is called, its contract can be easily retrieved: if `m` fulfills LSP we use Definition 4.4, otherwise `m` has only a single contract.

Definition 5.2. The body of a method $\delta.\mathtt{m}(\overline{\mathtt{p}})$ is in *shallow single original normal form*[3] (SSONF) if it is of the form

$$\{\mathtt{S}_1\,;\ \textbf{original}(\overline{\mathtt{p}})\,;\ \mathtt{S}_2\,;\} \tag{1}$$

where $\mathtt{S}_1$ and $\mathtt{S}_2$ are BP-sequences.

By wrapping **original** between two sequences whose behavior is not affected by delta applications, we make specification and verification local to deltas.

Some, but not all methods can be refactored to SSONF. In Listing 2 is a method `m` and its refactoring to SSONF, while in Listing 3 we show three methods that cannot be refactored to SSONF. Methods `m1` and `m2` cannot be refactored in SSONF when **original** and `m` are assumed to be not BP-methods. Whether **original** is executed in `m3` depends on the condition `x > 0`, therefore `m3` cannot be refactored to SSONF.

5.2 Add New Behavior Principle

An important limitation of LSP is that a modifying delta cannot extend the frame of its contract, even when it assigns newly added fields. But it is obvious that when the body of a modifying delta is in SSONF, assignments to new fields will not change the behavior of any previous version of the method. We overcome this restriction with the *Add New Behavior Principle*.

[3] SSNOF can be easily extended to cover modifications of a method `m` refactorable to $\{\mathtt{S}_1\,;\ \mathtt{T\ x\ =}\ \textbf{original}(\overline{\mathtt{p}})\,;\ \mathtt{S}_2\,;\}$. We omit this for brevity.

```
modifies Unit m1() {
  original(); // non BP
  original(); // non BP
}

modifies Unit m2() {
  original(); // non BP
  m(); // non BP-method
}
```

```
modifies Unit m3(Int x)
{
  if (x > 0) {
    original(x);
  } else {
    field1 += x;
  }
}
```

Listing 3. Modifying deltas not refactorable to SSONF

Definition 5.3. Let $\{\mathtt{f}_1, \ldots, \mathtt{f}_N\}$ be the fields δ adds to class `C`. We say δ modifies `m` according to the *Add New Behavior Principle* (ANBP) when $\delta.\mathtt{m}$ is in SSONF form (1) and $\mathtt{S}_i.\mathtt{frame} \subseteq \{\mathtt{f}_1, \ldots, \mathtt{f}_N\}$ for $i = 1, 2$. We also say that δ *adds new behavior* to `m`, that $\delta.\mathtt{m}$ satisfies the ANBP, and that the $\mathtt{S}_i$ *add new behavior*.

Since $(\mathtt{m.r}, \mathtt{m.e}, \mathtt{m.frame}) \not\geq (\delta.\mathtt{m.r}, \delta.\mathtt{m.e}, \delta.\mathtt{m.frame})$ follows from $\delta.\mathtt{m.frame} \not\subseteq \mathtt{m.frame}$, we are able to break LSP by applying ANBP. Listing 4 shows an example of a modifying delta satisfying the ANBP.

```
modifies C {
  adds Int field1;
  modifies Unit m() {
    original();
    field1 +=1;
  }
}
```

Listing 4. A modifying delta satisfying the ANBP

Lemma 5.4. *Let* `m` *be a method,* `m.r` *and* `m.e` *its Boolean JML contract clauses without references to non BP-methods, and assume* δ *adds new behavior to* `m`*. Let* `S` *be any sequence of statements added by* δ*. Then* `m.r` *and* `m.e` *are invariants of* `S`*, that is, if* `S` *is executed in any state satisfying* `m.r` *(*`m.e`*) and it terminates, then* `m.r` *(*`m.e`*) holds again in the final state.*

This lemma (the proof is obvious) states that no part of an existing contract can be invalidated by adding new behavior, as long as at most BP-methods are referenced in it. The restriction to non BP-methods is necessary: Assume the contract of $\mathtt{m}_1$ refers to the value returned by method $\mathtt{m}_2$ and δ adds as new behavior to $\mathtt{m}_1$ an assignment to the newly added field `f` before the call to **original**. In addition, δ modifies $\mathtt{m}_2$ by assigning to `f` and the returned value depends on the value of `f`. In this case, we are not able to assert that δ preserves the original behavior of $\mathtt{m}_1$.

In general, referring to a non BP-method in a specification, or calling it, leads to changed behavior and impedes compositional verification.

Using Lemma 5.4 we can easily extend the concept of Liskov-modular verification by adding new behavior. In light of Theorem 5.8, we refrain from giving the details.

5.3 Conjunctive Precondition Principle

Lemma 5.4 implies that new behavior added by a delta to a method `m` does not affect the validity of its precondition

`m.r` and postcondition `m.e`. If `m.r` is `True`, then we need not be concerned about the expected behavior of `m`. In contrast, if `m.r` is non-trivial, we have to prove that before calling `m` its precondition `m.r` holds. This is always the case when δ`.m.r` $\Rightarrow$ `m.r`. The authors of [41] propose an implicit feature-oriented contract composition mechanism called *conjunctive contract refinement*. With this mechanism, contracts are always *conjunctively refined*, i.e., modified by adding new pre- and postconditions conjunctively. With the ANBP, old contracts are preserved, but since in addition we require the old precondition to hold, we follow the idea behind conjunctive contract refinement by defining the following principle.

Definition 5.5. Let δ modify method `m`. We say that δ *satisfies* the *Conjunctive Precondition Principle* (CPP) if its precondition implies the precondition of each version of `m` to which δ is applicable.

This principle can be enforced for an existing specification, for example, by propagating preconditions along the delta application order. By automating this procedure it would be possible to avoid having to specify additional preconditions by hand. This is important to lower verification effort.

5.4 Change Old Behavior Principle

The main limitation of ANBP is its built-in behavioral monotonicity: the inability to override existing behavior goes against the grain of DOP. Thus, we generalize this principle:

Definition 5.6. Let δ be a delta modifying `C.m(`$\overline{\text{p}}$`)`. We say that δ satisfies the *Change Old Behavior Principle* (COBP) when exactly only one of the following cases applies:

1. δ`.C.m` is in SSONF {S_1; **original**(`p`); S_2;}, where S_1 *adds new behavior*;
2. δ`.C.m` is a BP-sequence.

By requiring methods to be in SSONF, COBP enforces only an already natural pattern for the design of deltas, namely, "allocate and set new fields—call to **original**—possibly override old values and result". This ensures that the behavior of the call to **original** is not unaffected by the delta's changes, and permits to apply locally-verifiable effective changes on top. Listing 5 shows an example of a modification fulfilling COBP.

```
1 modifies C {
2   modifies Unit m () {
3     original();
4     field += 1;
5   }
6 }
```

Listing 5. Example of a modification fulfilling the COBP

The COBP might still seem restrictive, but in fact it can be generalized to the following, fairly common situation:

{S_1; `if (boolExpr)` {**original**(`p`)}; S_2;} ,

where the execution of **original** is conditioned with a Boolean expression. When **original** is not executed, this is equivalent to a BP-sequence. Provided that `boolExpr` does not contain calls to non BP-methods, COBP covers the other case. For brevity we omit formal details.

Obviously, the following fact about ANBP and COBP holds:

Lemma 5.7. *If a delta δ satisfies ANBP, then it satisfies COBP.*

Hence, for correctness it suffices to focus on the COBP. Assuming LSP and COBP (modularly combined) we can prove, with AE, the correct behavior of each possible product by verifying each delta in isolation. The correctness of this family-oriented verification approach is formally stated in the following theorem:

Theorem 5.8. *Given a delta model consisting of a core C and a set of deltas D, assume the following holds:*

(1) The body of each method `m` in C is a BP-sequence, and `m` satisfies its contract.
(2) For all δ occuring in D and for all methods `m` in δ one of the following conditions holds:
  a. `m` is modified by δ, it fulfills the COBP, and satisfies its contract under the CPP.
  b. `m` is a BP-method, is modified by δ and it fulfills the LSP and is LSP-verified.
  c. `m` is added by δ, its body is a BP-sequence and it satisfies its contract.
(3) No contract can call a non BP-method.

Then each variant that can be generated from the given delta model satisfies its specification, i.e., each of its methods satisfies its contract.

Proof. The proof is by induction on the length k of a sequence of delta applications that results in a valid variant. We prove that each such sequence results in a variant that satisfies its specification.

The base case $k = 0$ follows from assumption (1). For the step case, let $\Delta = \delta_{P_1} \cdots \delta_{P_k}$ be a valid sequence of deltas such that $P = C\,\Delta$ is the variant generated by applying Δ to C. Assume P satisfies its specification. Let $\delta_{P_{k+1}} \in D$ be a valid delta applicable to P and $P' = P\,\delta_{P_{k+1}}$ the resulting product. Also, let $\mathtt{m}_i$ denote the version of a method `m` (if existing) after the application of δ_{P_i}, where $0 < i \leq k$ and $\mathtt{m}_0 = C.\mathtt{m}$. If $\mathtt{m}_i$ is not modified or removed by $\delta_{P_{i+1}}$, then $\mathtt{m}_{i+1} = \mathtt{m}_i$, $\mathtt{m}_{i+1=i}$ for short. We prove that each method in P' satisfies its contract.

By assumption (2) we know that methods added or modified by $\delta_{P_{k+1}}$ satisfy their contracts, therefore we have to prove that each method $\mathtt{m}_{k+1=k}$ in P still satisfies its contract. By assumption (3) the validity of any contract of these methods cannot possibly be affected by the delta application.

We prove that each method $\mathtt{m}_{k+1=k}$ satisfies its contract when its body is a BP-sequence as well when it is not. In the first case, since the behavior of a BP-sequence cannot be affected by subsequent delta application, we have that: (i) if $\mathtt{m}_{k+1=k} = \mathtt{m}_0$ then (1) holds, (ii) if $\mathtt{m}_{k+1=k} = \mathtt{m}_i$, with $0 < i \leq k$,

```
1  //before refactoring
2  modifies Unit m(p) {
3
4    if ( p1 == 0 ) {
5      field1 = 1;
6      m1(p1); //BP-method
7      original(p);
8      field2 +=1;
9    } else {
10     field1 = 2;
11     original(p);
12     m2(p2); //BP-method
13     field2 +=2;
14   }
15 }
16
```

```
1  //after refactoring
2  modifies Unit m(p) {
3    Bool cond = p1 == 0;
4    if ( cond ) {
5      field1 = 1;
6      m1(p1); //BP-method
7    } else {
8      field1 = 2;
9    }
10   original(p);
11   if (cond) {
12     field2 +=1;
13   } else {
14     m2(p2); //BP-method
15     field2 +=2;
16   }
17 }
```

Listing 6. Refactoring a modification of `m` with parameters $\overline{p}$ = `Int p1, Int p2` (left) to normal form (right).

and m_{i-1} does not exist, then $m_{k+1=k}$ is added by δ_{P_i} and (2)c holds; (iii) otherwise, if $m_{k+1=k} = m_i \neq m_{i-1}$, with $0 < i \leq k$, then it is modified by δ_{P_i}, and if $m_{k+1=k}$ is a BP-method then (2)b holds, otherwise (2)a holds. Therefore, each method $m_{k+1=k}$ whose body is a BP-sequence still satisfies its contract after the application of $\delta_{P_{k+1}}$.

We turn to method $m_{k+1=k}$ whose body is not a BP-sequence, i.e., it contains a call to a non BP-method. Since the only allowed call to a non BP-method is **original**, $m_{k+1=k}$ must result from the application of a delta δ_{P_i}, $0 < i \leq k$ that calls **original**. This $\delta_{P_i}.m$ is in SSONF $\{S_1\,;\,$ **original**$(\overline{p})\,;\,S_2\,;\}$ and it must fulfill assumption (2)a. Because the behavior of **original** and of BP-sequences S_i are not affected by subsequent delta applications, the assumption still holds. Since each method $m_{k+1} \neq m_k$ and $m_{k+1=k}$ satisfies its respective contract, all methods in P' do. □

Invariants. Concerning object invariants, we use the same approach as [21]: no addition, modification or deletion of invariants can be specified in deltas, unless the invariant a delta adds refers only to newly added fields.

6 Achieving Normal Form

In many cases, a program not in SSONF is equivalent to a program in SSONF. For example, the program shown in Listing 6 on the left has the same behavior as the one on the right. Corresponding segments on the left and right are highlighted with the same color. The crucial observation is that both programs can be related with a series of simple transformation steps. The correctness of these transformations is ensured by AE, which makes it possible to verify a schematic transformation rule for all possible instances.

```
if ( e^boolean(frameE :≈ footprintE) ) {
  Q1(frameQ1 :≈ footprintQ1);
} else {
  Q2(frameQ2 :≈ footprintQ2);
}
```

↓ **CSCE**

```
/*@ ae_constraint
  @ x ∉ frameQ1 && x ∉ footprintQ1 &&
  @ x ∉ frameQ2 && x ∉ footprintQ2;
  @*/
x = e^boolean(frameE :≈ footprintE);
if (x) {
  Q1(frameQ1 :≈ footprintQ1);
} else {
  Q2(frameQ2 :≈ footprintQ2);
}
```

Listing 7. Formalization of rule CSCE in the AE framework

We employ the following transformation rules that are also sufficient to transform the program in Listing 6.

1. **Conditional Statement Condition Extraction (CSCE)**: This rule extracts an abstract Boolean condition (in Listing 6 on the left in Line 4) and replaces it with a variable (in the example `cond`) that captures its value and that is not modifiable by any subsequent statements.
   The formalization of rule CSCE in the AE framework of Section 2 is given in Listing 7. For brevity, we omit the formalization of the remaining rules.
2. **Unfold If Branch Prefix**: Splits the **if**-branch of a conditional into two consecutive **if**-statements, distributing their bodies.
3. **Unfold Else Branch Prefix**: As above, but **else**-branch.
4. **Consolidate Duplicate Conditional Fragments Extract Prefix** [15]: This rule allows us to extract the same sequence of statements, or *prefix*, from the **if**-branch and the **else**-branch of a conditional, with the constraint that none of the extracted statements may affect the value of the conditional expression.
5. **Split Conditional Statement**: Splits a conditional statement in two conditional statements having complementary guards. In our example we use its inverse.

7 Case Studies

We demonstrate the feasibility of our approach and show that it is *expressive enough* to specify existing software product lines. Section 7.1 discusses the product line **SimpleSPL** that was specifically designed to guide and validate our approach, with a focus on the application of the presented principles. Section 7.2 demonstrates that our approach is applicable to non-trivial *legacy* software product lines using two SPLs widely used in the literature. In Section 7.3 we perform an

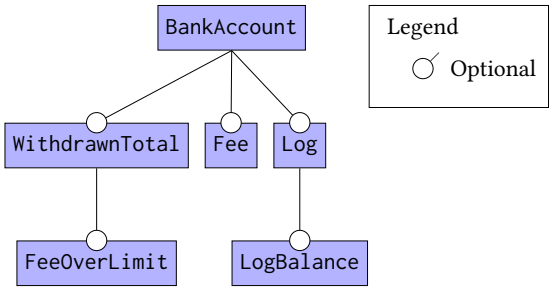

Figure 1. Feature diagram of **SimpleSPL**

```
1 delta DWithdrawnTotal when WithdrawnTotal {
2  modifies class BankAccount {
3   adds Int withdrawnTotal;
4   //@ ensures (x < 0) ==>
5   //@          (withdrawnTotal == \old(withdrawnTotal) - x);
6   modifies public Unit update(Int x) { // satisfies ANBP
7     if (x < 0) withdrawnTotal = withdrawnTotal - x;
8     original(x);
9   }
10  }
11 }
```

Listing 8. Delta `DWithdrawnTotal`

experimental evaluation against a product-based approach. The case studies and the used KeY system can be found at https://github.com/se-tud/GPCE21-case-studies.

7.1 SimpleSPL

Figure 1 shows the feature model of **SimpleSPL**. The SPL consists of one class `BankAccount`, one method `update` (Listing 1), five deltas, and 18 possible products. Each delta modifies method `update` by adding new or changing old behavior.

7.1.1 Add New Behavior Principle. In this product line, the deltas `DWithdrawnTotal` (shown in Listing 8) and `DLog` (omitted for brevity) satisfy the ANBP: The body of `update` in `DWithdrawnTotal` is in SSONF and its frame is expanded by assigning `withdrawnTotal`, a newly added field.

7.1.2 Change Old Behavior Principle. In Listing 9 we show the specification of delta `DFee` triggered by the selection of feature `Fee`. Delta `DFee` modifies `update` by decreasing `balance` by the value of `fee`. Variable `balance` might have been assigned in **original**, therefore, in the contract we need to refer to the value it had after the call to **original**. The JML extension mentioned in Section 2.3 permits to achieve this with the label `l` in `\old(this.balance,l)` (Line 4).

Using the LSP alone [21], it would not be possible to specify the precise contracts for this modifying delta. Since the method satisfies COBP we do not have to worry about the contracts of previous versions referenced by **original**.

```
1 delta DFee when Fee {
2  modifies class BankAccount {
3   adds Int fee;
4   //@ ensures (x < 0) ==> (this.balance == \old(this.balance,l) - fee);
5   modifies Unit update(Int x) { // satisfies COBP
6     l: original(x);
7     if (x < 0) balance = balance - fee;
8   }
9  }
10 }
```

Listing 9. Delta `DFee` with specified contract for modification of method `update`

7.2 Coverage

We continue with an investigation of the coverage of existing product lines. Because of the scarcity of suitable case studies in ABS, we opted to remodel two existing product lines `BankAccountSPL` [42] and `MinePumpPL` [5] in ABS using the DOP paradigm. Originally these models were designed with *feature-oriented programming* (FOP) [6, 34] in Java, whose implementations are available at SPL2go [40]. Both implement interesting behavioral variability. Only the latter required any substantial remodeling, due to method modifications not fulfilling COBP and LSP. This allowed us to evaluate our approach for an SPL on which it would not be applicable originally.

7.2.1 Remodeling. The remodeling required to address some (non-essential for the case study) differences between Java and ABS and, of course, to take into account the constraints imposed by our approach.

We opted for a plain remodeling of the Java classes as ABS classes. Since DOP was designed to *extend* FOP, a straightforward remodeling consists of mapping the base feature to the core and any other feature `F` to a delta `DF`: the core is the base variant and each delta `DF` contains the changes to be applied when `F` is selected.

Due to strong object encapsulation in ABS [23], direct references to object fields of foreign classes are unsupported and were remodeled in the code as well as in contracts by calls to *getter/setter* methods exposed in interfaces (which were implemented, if necessary). In DOP the granularity of a modification is at the method level, therefore, we remodeled modifiable final fields as access methods satisfying LSP. In the following we describe both case studies in greater detail.

7.2.2 BankAccountSPL. `BankAccountSPL` [42] is a well-known case study implementing a small system for the bank account management. We chose it because of its similarities to `SimpleSPL`, but its considerably higher complexity due to the presence of invariants, two classes, and a larger amount of modified methods. `BankAccountSPL`, whose feature model is shown in Figure 2, consists of six features and 24 products. The classes are `Account` and `Application`. The former handles withdrawal, deposit, interest, and daily limit for withdrawals.

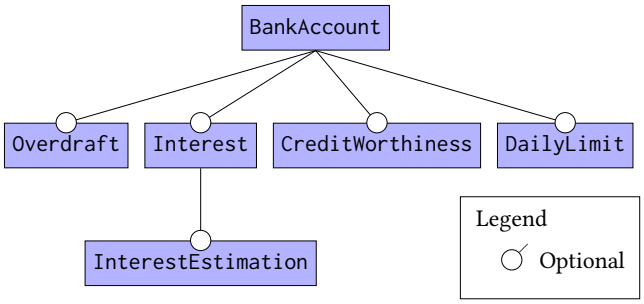

Figure 2. Feature diagram of BankAccountSPL

The latter may restore a daily limit and update the balance of Account according to the given interest rate.

The behavior of the FOP version comes with JML contracts, therefore, we relied on these contracts to verify the correctness of the product line. We observe that (i) **original** is the only non BP-method called, (ii) all modified methods calling **original** are already in SSONF, and (iii) these methods satisfy COBP. Hence, BankAccountSPL is fully covered by our approach without any need of structural remodeling.

The only issues we met were the contracts of method Application.nextYear and Application.nextDay in DeltaInterest: the specified behavior of the former assumed that the original call does not change the value of existing fields, and since the only previous version of the method is empty we removed this original call. For the latter, the problem was related to remodeling: since we wrapped all references to object fields with getter and setter methods, we had to choose between over-specifying the contracts to have very precise assignable clauses, or to use labeled **original** calls and leave the additional work to the verification tool. We opted for the second solution.

7.2.3 MinePumpPL. The MinePumpPL system [5], based on work in the CONIC project [29], simulates a water pump in a mining operation. If the pump is running it keeps the mining shaft dry and safety precautions should be applied: if the presence of methane is detected, the pump should stop running to avoid a risk of explosion.

The feature model of MinePumpPL (see Figure 3) consists of one abstract and nine concrete features and 64 valid products. The model for the FOP version of this SPL is refactored into an equivalent one with WaterSensor as its base feature, and all the leaves as optional features, i.e., with seven concrete features. According to our remodeling strategy, each delta is triggered by the selection of two features, one leaf and its parent: for example, DeltaHighWaterSensor is triggered by the selection of WaterSensor and High. This system consists of two classes, MinePump and Environment: the former implements the simulation of the pump, while the latter permits to know and to modify the status of the environment in which the pump is deployed.

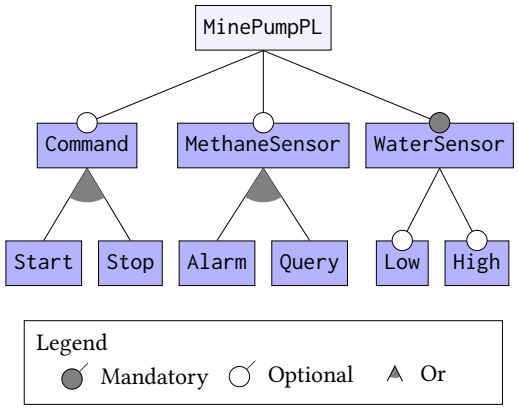

Figure 3. Feature diagram of MinePumpPL

```
1  class MinePump {
2    Bool pumpRunning = false;
3    Unit activatePump() { pumpRunning = True; }
4  }
5
6  delta DeltaMethaneSensorQuery {
7    modifies MinePump {
8      modifies Unit activatePump() {
9        if (!isMethaneAlarm()) original();
10     }
11   }
12 }
13
14 delta DeltaHighWaterSensor {
15   modifies class MinePump {
16     modifies Unit processEnvironment() {
17       if (!pumpRunning && isHighWaterLevel()) {
18         activatePump();
19         original();
20       } else {
21         original();
22       }
23     }
24   }
25 }
```

Listing 10. Code of DeltaMethaneSensorQuery, parts of the core, and DeltaHighWaterSensor before remodeling

A contract-based specification for the behavior of the FOP version of MinePumpPL was missing, so we added it in the DOP version. Due to the presence of calls to non BP-methods, additional remodeling is needed. We show one example in Listing 10: activatePump, declared in the core at Line 3, turns the pump on, is modified by DeltaMethaneSensorQuery (Lines 8–10), and called by processEnvironment in DeltaHighWaterSensor (Line 18) (this delta can be refactored to obtain a single call to **original**). If activatePump does not satisfy LSP then our approach cannot be applied. Unfortunately, making activatePump do so is not a good option, because the behavior of activatePump is too specific and crucial for the system to admit a more general contract.

Therefore, as shown in Listing 11, we introduce a safety check on activatePump in the core at Line 4, where a newly

```
1  class MinePump {
2    //@ ensures (problemCount() == 0) ==> pumpRunning;
3    Unit activatePump() {
4      if (problemCount() == 0)
5        pumpRunning = True;
6    }
7
8    //@ ensures \result >= 0;
9    Int problemCount() { return 0; }
10 }
11
12 delta DeltaMethaneSensorQuery {
13   modifies MinePump {
14     //@ ensures \result >= 0 && isMethaneAlarm() ==> \result >= 1;
15     Int problemCount() {
16       Int count = original();
17       if (isMethaneAlarm())
18         count = count + 1;
19       return count;
20     }
21   }
22 }
```

Listing 11. Remodeled core and `DeltaMethaneSensorQuery` with JML specification

```
1  class MinePump {
2    Unit timeShift() {
3      if (pumpRunning)
4        env.lowerWaterLevel();
5      if (systemActive)
6        processEnvironment();
7    }
8
9    Unit processEnvironment {}
10 }
```

Listing 12. Method `timeShift` in the core

created access method `problemCount` (Line 9) is called. Furthermore, we replaced the modification of `activatePump` in `DeltaMethaneSensorQuery` with the one of `problemCount` (Lines 15–20). The behavior of the latter can be sufficiently weakened using "*result* >= ..." to make it satisfy LSP.

Listing 12 shows another problem we encountered: method `timeShift`, simulating the passing of time, never modified, calls `processEnvironment` (Line 6), modified by deltas `DeltaHighWaterSensor`, `DeltaLowWaterSensor`. Also here, generalizing the contract is not a good solution. Since `processEnviroment` has no function in the core, we decided to remove it, and to replace its modifications in `DeltaHighWaterSensor` and `DeltaLowWaterSensor` with modifications of method `timeShift` fulfilling COBP, behaving identically. For brevity we do not show the code of these changes.

Planning and performing the transformations took roughly 4 hours, while the formal specification and deductive verification itself took approximately 14 hours.

7.3 Experimental Evaluation

To evaluate our approach experimentally, we compare it to product-based analysis. We consider the cost of verifying method `timeShift` for a subset of `MinePumpPL` (the most complex problem in our case studies) consisting of three features `Low`, `High`, `Alarm`, as well as seven products. We omit the effort for verification of `timeShift` in the core, because it is verified once in both approaches. Since each feature corresponds one-to-one to a delta that modifies `timeshift`, the number of methods to be verified in our approach is three. In contrast, with the product-based approach we have to verify seven methods.

Table 1. Cost of verification approaches

Delta	Nodes
Low	5772
High	6254
Alarm	1689
Total	13715

(a) Delta-based verification

Product	Nodes
Low (L)	2233
High (H)	2472
Alarm (A)	1092
L+H	2517
L+A	1860
H+A	1869
L+H+A	1883
Total	13926

(b) Product-based verification

Tables 1a and 1b break down the cost for the delta- and product-based approach, respectively, in terms of nodes of the resulting proof trees (all proofs are fully automatic).

In this initial scenario with a small number of (three) features, our approach performs comparable to the product-based approach. The product-based approach, however, does not scale when the number of features increases, because in the worst case the number of proofs is exponential in the number of features; and is outperformed by our approach, which is suitable for evolving SPLs.

To demonstrate the scaling effect, we present an evolution scenario for `MinePumpPL`. We add a new delta `DeltaEmergency` triggered by selecting a new feature `Emergency`. Delta `DeltaEmergency` modifies method `timeShift` by shutting down the pump if an emergency occurs at the end of the shift. Selection of `Emergency` is not constrained, increasing the number of products from 7 to 15, and the modification of `DeltaEmergency` is always the final one applied for each product variant. With our approach the additional effort to verify correctness of all eight new products consists of 4598 proof nodes, because we only need to verify `timeShift` for `DeltaEmergency`. In contrast, individually verifying each of the eight new products requires 19107 proof nodes. Clearly, only very few features are needed to outperform the product-based approach. In fact, with four features, the total amount of proof nodes is 18313 and 33033 for the delta- and product-based approach, respectively.

8 Related Work

Composition. In [42] a family-based approach is presented that composes specification and implementation of all variants into a metaproduct. Its monolithic nature requires the

metaproduct to be recreated and reverified after each change. Composition of partial proofs for features and deltas into proofs for all products is proposed in [12–14, 18, 43].

Thüm et al. [41] present a set of mechanisms to define and *compose contracts* for feature-oriented method refinements. The principles we introduced share aspects with some of these mechanisms. As noted in Section 5, the *Add New Behavior Principle* combined with the *Conjunctive Precondition Principle* follows the same idea as *conjunctive contract refinement*: Modifications that add new behavior ensure both the old and new postcondition, but must require both the old and new precondition. Also, the behavior specified in a contract of a modification satisfying the *Change Old Behavior Principle* (partially) overrides previous behavior, similar to their *contract overriding*. The decisive difference to our proposal is that we combine the contract composition principles with a normal form for method modification in deltas. This enables the composition Theorem 5.8 *despite* contract overriding.

Abstract Contracts. Knüppel et al. [27] propose *FEFALUTION*, a feature-family-based algorithm for the verification of evolving product lines. They aim to reduce the overall verification cost by using *pure abstract contracts* [8, 22] to verify features and their interaction. During a feature-based phase, using *abstract contracts*, FEFALUTION generates fully precise behavioral specifications for methods without a unique behavior in terms of partial proofs for the correctness of the contracts of these methods. During a second, family-based phase, a meta product representing the full product family is generated and it is used by FEFALUTION when it attempts to complete all partial proofs.

Our approach neither has a feature- nor a family-based phase. Instead, the main tasks are to check whether the requirements of Theorem 5.8 are satisfied and then to verify each core method and each delta modification in isolation. The approach of Knüppel et al. [27] does not limit the form of the refinements and does not require to satisfy any restrictive principle, however, the generality of fully abstract contracts leads to inefficiencies during construction of the partial proofs and when constructing the full proofs.

Abstract Execution. Abstract Execution has been applied in a number of different contexts: (i) Deriving preconditions for the safe application of refactoring rules [38], (ii) analyzing the cost impact of program transformation rules [3], (iii) enabling of code parallelization [19], (iv) "Correct-by–Construction" program development [45], and (v) the correctness of rule-based compilation [37]. The application of AE to the modular verification of software product lines has, to the best of our knowledge, not been studied before.

9 Discussion and Future Work

The main innovative aspect of our approach is to enable liberal contract composition principles by imposing SSONF on method modifications in deltas. Thereby, we render the specification contracts and, hence, the verification local (Theorem 5.8). This has important consequences: first, it makes it easier to write specifications, because different versions of a method can be specified independently of each other, thus alleviating the specification bottleneck. Second, one can create more precise specifications than under an LSP-regime. Third, and most important, it is sufficient to verify all core and delta contracts *locally*, after checking that the COBP conditions apply. For this reason, we call our approach *delta-based*. In particular, one does not need to actually generate any variant to verify its correctness.

The necessity of a normal form potentially is also the main drawback of our approach, because it limits the code design space. We argue that in practice the limitation is not very severe: first, the restriction embodied in the COBP (Definition 5.6) is quite natural for the design of modifying deltas and follows the pattern "allocate and set new fields—call to **original**—possibly override old values and result". Complications arise, when the call to **original** is not *shallow*. But this is not a show-stopper, because we can use program transformation to generate behaviorally equivalent code in SSONF in most cases. We did not encounter any natural example that would have excluded SSONF.

Of course, code transformation requires effort and expertise and, obviously, it deviates from the original code. The latter we deem unproblematic: The transformed code is only required *for verification*, because the original code is behaviorally equivalent. The former issue clearly constitutes an opportunity for more far-reaching user support and automation. For example, it should be possible to identify *code transformation patterns* that systematically bring **original** calls to the top-level ("shallow") and move overriding assignments to a permitted position. This is a topic for future work.

We stress that our whole approach hinges on a recent innovation in static program verification: abstract execution. It allows us not only to prove behavioral equivalence of the transformation schemata, but it is instrumental to prove correctness of a delta modification *without the need to know the concrete instance of* **original**, but merely by relying on the constraints on its dynamic frame.

The AE specification language and implementation supports abstract symbols that can be instantiated to abruptly terminating code. To keep the presentation simple we did not use this. It would be technical, but conceptually straightforward, to integrate abrupt termination into our approach.

For the reasons given in Section 3, we based our account on the ABS language. As soon as the overhaul of DOP for Java is completed, it would make sense to integrate our approach into that tool chain.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel,

Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 2014. The KeY Platform for Verification and Analysis of Java Programs. In *Verified Software: Theories, Tools and Experiments (LNCS, Vol. 8471)*, Dimitra Giannakopoulou and Daniel Kroening (Eds.). Springer, Cham, 55–71. https://doi.org/10.1007/978-3-319-12154-3_4

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS, Vol. 10001. Springer, Cham. https://doi.org/10.1007/978-3-319-49812-6

[3] Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. 2021. Certified Abstract Cost Analysis. In *Fundamental Approaches to Software Engineering (LNCS, Vol. 12649)*, Esther Guerra and Mariëlle Stoelinga (Eds.). Springer, Cham, 24–45. https://doi.org/10.1007/978-3-030-71500-7_2

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

[5] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Intl. Conf. on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Los Alamitos, CA, 482–491. https://doi.org/10.1109/ICSE.2013.6606594

[6] D. Batory, J.N. Sarvela, and A. Rauschmayer. 2004. Scaling step-wise refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23

[7] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2012. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In *Proc. 7th Conf. on Systems Software Verification (SSV) (EPTCS, Vol. 102)*, Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich (Eds.). Open Publishing Association, Ithaca, NY, 18–32. https://doi.org/10.4204/EPTCS.102.4

[8] Richard Bubel, Reiner Hähnle, and Maria Pelevina. 2014. Fully Abstract Operation Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications (LNCS, Vol. 8803)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Berlin, Heidelberg, 120–134. https://doi.org/10.1007/978-3-662-45231-8_9

[9] Richard Bubel, Andreas Roth, and Philipp Rümmer. 2008. Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic. *Electr. Notes Theor. Comput. Sci.* 199 (2008), 107–128. https://doi.org/10.1016/j.entcs.2007.11.015

[10] Rod M. Burstall. 1974. Program proving as hand simulation with a little induction. In *Information Processing '74*. Elsevier/North-Holland, Amsterdam, 308–312.

[11] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. 2011. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Formal Methods for Eternal Networked Software Systems: 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM, Bertinoro, Italy (LNCS, Vol. 6659)*, Marco Bernardo and Valérie Issarny (Eds.). Springer, Berlin, Heidelberg, 417–457. https://doi.org/10.1007/978-3-642-21455-4_13

[12] Ferruccio Damiani, Olaf Owe, Johan Dovland, Ina Schaefer, Einar Broch Johnsen, and Ingrid Chieh Yu. 2012. A transformational proof system for delta-oriented programming. In *16th International Software Product Line Conference, SPLC'12* (Salvador, Brasil), Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides (Eds.). ACM, New York, NY, USA, 53–60. https://doi.org/10.1145/2364412.2364422

[13] Benjamin Delaware, William R. Cook, and Don S. Batory. 2011. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2048066.2048113

[14] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/2429069.2429094

[15] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston.

[16] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston. 2nd edition.

[17] Benny Godlin and Ofer Strichman. 2013. Regression Verification: Proving the Equivalence of Similar Programs. *Softw. Test., Verif. Reliab.* 23, 3 (2013), 241–258. https://doi.org/10.1002/stvr.1472

[18] Ali Gondal, Michael Poppleton, and Michael J. Butler. 2011. Composing Event-B Specifications - Case-Study Experience. In *Software Composition - 10th International Conference, SC@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings (LNCS, Vol. 6708)*, Sven Apel and Ethan K. Jackson (Eds.). Springer, Berlin, Heidelberg, 100–115. https://doi.org/10.1007/978-3-642-22045-6_7

[19] Reiner Hähnle, Asmae Heydari Tabar, Arya Mazaheri, Mohammad Norouzi, Dominic Steinhöfel, and Felix Wolf. 2020. Safer Parallelization. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles (LNCS, Vol. 12477)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Cham, 117–137. https://doi.org/10.1007/978-3-030-61470-6_8

[20] Reiner Hähnle and Marieke Huisman. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science: State of the Art and Perspectives*, Bernhard Steffen and Gerhard Woeginger (Eds.). LNCS, Vol. 10000. Springer, Cham, 345–373. https://doi.org/10.1007/978-3-319-91908-9_18

[21] Reiner Hähnle and Ina Schaefer. 2012. A Liskov Principle for Delta-Oriented Programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (LNCS, Vol. 7609)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Berlin, Heidelberg, 32–46. https://doi.org/10.1007/978-3-642-34026-0_4

[22] Reiner Hähnle, Ina Schaefer, and Richard Bubel. 2013. Reuse in Software Verification by Abstract Method Calls. In *Automated Deduction – CADE-24 (LNCS, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, Berlin, Heidelberg, 300–314. https://doi.org/10.1007/978-3-642-38574-2_21

[23] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects (LNCS, Vol. 6957)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer, Berlin, Heidelberg, 142–164. https://doi.org/10.1007/978-3-642-25271-6_8

[24] K. C. Kang, Jaejoon Lee, and P. Donohoe. 2002. Feature-oriented product line engineering. *IEEE Software* 19, 4 (2002), 58–65. https://doi.org/10.1109/MS.2002.1020288

[25] Ioannis T. Kassios. 2011. The Dynamic Frames Theory. *Formal Asp. Comput.* 23, 3 (2011), 267–288. https://doi.org/10.1007/s00165-010-0152-5

[26] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[27] 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*. LNCS, Vol. 12345. Springer, Cham, 122–148. https://doi.org/10.1007/978-3-030-64354-6_5

[28] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. 2014. DeltaJ 1.5: Delta-oriented Programming for Java 1.5. In *Proc. Intl. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland). ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/2647508.2647512

[29] Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister. 1983. CONIC: An Integrated Approach to Distributed Computer Control Systems. *IEE Proceedings E (Computers and Digital Techniques)* 130, 1 (1983), 1–10. https://doi.org/10.1049/ip-e.1983.0001

[30] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. *JML Reference Manual*. University of Central Florida. http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf Draft revision 2344.

[31] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[32] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2018. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM* 61, 2 (2018), 84–91. https://doi.org/10.1145/3166064

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

[34] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 — Object-Oriented Programming (LNCS, Vol. 1241)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer, Berlin, Heidelberg, 419–443. https://doi.org/10.1007/BFb0053389

[35] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (LNCS, Vol. 6287)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, Berlin, Heidelberg, 77–91. https://doi.org/10.1007/978-3-642-15579-6_6

[36] Dominic Steinhöfel. 2020. REFINITY to Model and Prove Program Transformation Rules. In *Programming Languages and Systems (LNCS, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, Cham, 311–319. https://doi.org/10.1007/978-3-030-64437-6_16

[37] Dominic Steinhöfel and Reiner Hähnle. 2018. Modular, Correct Compilation with Automatic Soundness Proofs. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling (LNCS, Vol. 11244)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Cham, 424–447. https://doi.org/10.1007/978-3-030-03418-4_25

[38] Dominic Steinhöfel and Reiner Hähnle. 2019. Abstract Execution. In *Formal Methods – The Next 30 Years (LNCS, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, Cham, 319–336. https://doi.org/10.1007/978-3-030-30942-8_20

[39] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45. https://doi.org/10.1145/2580950

[40] Thomas Thüm and Fabian Benduhn. [n.d.]. SPL2go. http://spl2go.cs.ovgu.de. Accessed: 2021-09-23.

[41] Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. 2019. Feature-Oriented Contract Composition. In *Proc. 23rd Intl. Systems and Software Product Line Conference - Volume A* (Paris, France). ACM, New York, NY, USA, 25. https://doi.org/10.1145/3336294.3342374

[42] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-Based Deductive Verification of Software Product Lines. *SIGPLAN Not.* 48, 3 (Sept. 2012), 11–20. https://doi.org/10.1145/2480361.2371404

[43] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. 2011. Proof Composition for Deductive Verification of Software Product Lines. In *Fourth IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST, Berlin, Germany*. IEEE Computer Society, Los Alamitos, CA, 270–277. https://doi.org/10.1109/ICSTW.2011.48

[44] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, and Ina Schaefer. 2016. Parametric DeltaJ 1.5: Propagating Feature Attributes into Implementation Artifacts. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE), Wien (CEUR Workshop Proceedings, Vol. 1559)*. CEUR-WS.org, Aachen, 40–54. http://ceur-ws.org/Vol-1559/paper04.pdf

[45] David Winterland. 2020. *Abstract Execution for Correctness-by-Construction*. Master's thesis. Technische Universität Braunschweig. https://www.tu-braunschweig.de/isf/team/runge