Automatic refactoring of delta-oriented SPLs to remove-free form and replace-free form

International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automatic Refactoring of Delta-Oriented SPLs to Remove-free Form and Replace-free Form

Ferruccio Damiani · Michael Lienhardt · Luca Paolini

Received: 2018 / Accepted: 2018

Abstract Delta-Oriented Programming (DOP) is a flexible transformational approach to implement Software Product Lines (SPLs). In delta-oriented SPLs, variants are generated by applying operations contained in delta modules to a base program. These operations can add, remove or modify named elements in a program (e.g., classes, methods and fields in a Java program). This paper presents two notions of normal form for delta-oriented SPLs. Both normal forms do not contain the remove operation. Additionally, the second normal form enforces a limitation on the use of the method-modify operation. For each of the proposed normal forms an algorithm for refactoring a delta-oriented SPL into one that satisfies that normal form is described. The algorithms are formalized for a core calculus for delta-oriented SPLs of Java programs.

Keywords Delta-Oriented Programming · Featherweight Java · Software Product Lines · Automated Refactoring

This work has been partially supported by: EU Horizon 2020 project HyVar (`www.hyvar-project.eu`, GA No. 644298) and ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`).

Ferruccio Damiani
University of Turin, Turin, Italy
E-mail: ferruccio.damiani@unito.it

Michael Lienhardt
ONERA, Palaiseau, France
E-mail: michael.lienhardt@onera.fr

Luca Paolini
University of Turin, Turin, Italy
E-mail: luca.paolini@unito.it

1 Introduction

A *Software Product Line* (SPL) is a set of similar programs, called *variants*, that are generated from a common code base [12]. A flexible and modular transformational approach to implement SPLs is the *Delta-Oriented Programming* (DOP) [44] [3, Sect. 6.1.1]. A DOP product line comprises a *Feature Model* (FM), a *Configuration Knowledge* (CK), and an *Artifact Base* (AB). The FM provides an abstract description of variants in terms of *features* (each representing an abstract description of functionality): each variant is described by a set of features, called a *product*. The AB provides the (language-dependent) code artifacts used to build the variants, namely: a (possibly empty) base program and a set of *delta modules* (*deltas* for short) that are applied in sequence to the base program to transform it into a variant of the SPL. The CK provides a mapping from products to variants by describing the connection between the code artifacts in the AB and the features in the FM: it associates to each delta an *activation condition* over features and specifies an *application ordering* between deltas. Delta orientation allows for the automatic generation of its variants: once a product of the FM is selected, the deltas with an activation condition that are satisfied by the product are identified from the CK and are applied to the base program according to the application ordering in the CK to obtain the expected variant.

Each delta comprises *delta operations* that can *add*, *modify* or *remove* named elements in the base program (e.g., for Java programs, a delta can add, remove or modify class interfaces, fields and methods [34]). As pointed out in [45], thanks to this flexible operations, DOP supports *proactive* (i.e., planning all products in advance), *reactive* (i.e., developing an initial SPL com-

prising a limited set of products and evolving it as soon as new products are needed or new requirements arise), and *extractive* (i.e., gradually transforming a set of existing programs into an SPL) SPL development [36]. In particular, DOP is particularly suited for SPL evolution and extension, as modifying or adding variants can straightforwardly be achieved by adding to the SPL new deltas that modify, remove or add code on top of the existing variants of the SPL. However, as pointed out by Schulze et al. [46], a number of such SPL evolution and extension phases may introduce contradicting add and remove operations, leading to SPLs that are complex, and difficult to understand and to analyze.

Refactoring [22] is an established technique to reduce complexity and improve readability of programs. It consists of program transformations that change the internal structure of a program without altering its observable behaviour. In this paper, by refactoring a delta-oriented SPL we mean changing its FM, CK or AB without changing its products and variants [46,17,16].

In this paper, we present two notions of normal form for delta-oriented SPLs of Java programs: the *remove-free* form and the *replace-free* form. Both normal forms do not contain the remove operation. Additionally, the replace-free form enforces a limitation on the use of the method-modify operation. For each of the proposed normal forms an algorithm for refactoring a delta-oriented SPL into an SPL that satisfies that normal form is described. The SPLs produced by the refactoring algorithms satisfy further constraints: they have an empty base program and they contain only *atomic* deltas—a delta is atomic if it contains a single operation (e.g., it adds an empty class that extends `Object`, or it modifies a class by either changing its `extends`-clause, or by adding an attribute, or by modifying a method).

Actually, the refactoring of an SPL into atomic form (i.e., an SPL that has an empty base program and contains only atomic deltas) is performed as a preliminary step by both refactoring algorithms. This preliminary step simplifies the formulation of the refactoring algorithms, which refactor an atomic SPL into atomic *remove-free* form and into atomic *replace-free* form, respectively. Both refactoring algorithms (including the algorithm for refactoring into atomic form) leave the feature model of the SPL unchanged.

Both refactoring algorithms transform an SPL without requiring interaction with the developers of the SPL. However, as discussed in Section 6, in order to use refactoring in practice, it will be necessary to develop suitable tools that connect the AB of the refactored SPL to the AB of the original SPL and allow SPL developers to do a review pass on the refactored SPL, e.g., to merge some deltas that have the same activation condition and/or to reintroduce a non-empty base program.

We present the refactoring algorithms for *Imperative Featherweight Delta Java* (IFΔJ) [7], a core calculus for delta-oriented SPLs where variants are written in an imperative version of *Featherweight Java* (FJ) [27].

In previous work [16], we have already proposed two algorithms for refactoring IFΔJ SPLs into *remove-free* form and into *replace-free* form, respectively. These previous algorithms, which do not describe the preliminary refactoring of an SPL into atomic form, are quite complex to describe and understand. Instead, the refactoring algorithms presented in this paper provide a better understanding of the relations between the refactored SPL and the original SPL, thus paving the way towards the development of suitable tool support as advocated in Section 6.

Organization of the Paper. Section 2 recalls Imperative Featherweight Java (IFJ). Section 3 recalls delta-oriented SPLs by means of the IFΔJ language. Section 4 formalizes the notion of *atomic* IFΔJ SPL and describes a refactoring algorithm that converts any IFΔJ SPL into atomic form. Section 5 formalizes the notions of IFΔJ SPL in remove-free and in replace-free form, and describes the associated refactoring algorithms. Section 6 discusses the significance of the proposed refactoring algorithms. Section 7 discusses related work and Section 8 concludes the paper by outlining possible future work including also aspects concerning dynamic reconfiguration.

2 Imperative Featherweight Java

The abstract syntax of IFJ *programs* is given in Figure 1—explanations are given in the caption. Following Igarashi et al. [27], we use the overline notation for (possibly empty) sequences of elements: for instance $\overline{e}$ stands for a sequence of expressions $e_1, \ldots, e_n$ ($n \geq 0$). The empty sequence is denoted by $\emptyset$. Moreover, when no confusion may arise, we identify sequences of pairwise distinct elements with sets, e.g., we write $\overline{e}$ as short for $\{e_1, \ldots, e_n\}$. As usual, we identify the textual representations of IFJ programs modulo: (i) the order of class declarations or attribute declarations, and (ii) renaming of the formal parameters of methods. The following notational convention entails the assumption that classes declared in a program, attributes declared in a class, and formal parameters declared in a method have distinct names.

Notation 1 (Convention on sequences of named declarations) Whenever we write a sequence of named

P	$::= \overline{CD}$	Program
CD	$::=$ **class** `C` **extends** `C` { $\overline{AD}$ }	Class Declaration
AD	$::= FD \mid MD$	Attribute (Field or Method) Declaration
FD	$::=$ `C f`	Field Declaration
MH	$::=$ `C m`$(\overline{\texttt{C x}})$	Method Header
MD	$::= MH$ {**return** e;}	Method Declaration
e	$::=$ `x` $\mid$ e`.f` $\mid$ e`.m`$(\overline{e})$ $\mid$ **new** `C()` $\mid$ `(C)`e $\mid$ e`.f` $= e$ $\mid$ **null**	Expression

Fig. 1: **IFJ programs.** A program P is a sequence of class declarations $\overline{CD}$. A class declaration comprises the name `C` of the class, the name of the superclass (which must always be specified, even if it is the built-in class `Object`), and a list of attribute (field or method) declarations $\overline{AD}$. Variables `x` include the special variable `this` (implicitly bound in any method declaration MD), which may not be used as the name of a method's formal parameter. All fields and methods are public, there is no field shadowing, there is no method overloading, and each class is assumed to have an implicit constructor that initializes all fields to **null**.
An *attribute name* `a` is either a field name `f` or a method name `m`. Given a class declaration CD we write $\mathrm{dom}(CD)$ to denote the set of attribute names declared in CD. Given a program P, a class name `C` and an attribute name `a`, we write $\mathrm{dom}(P)$, $<:_P$, $\texttt{C}_P$ and $lookup_P(\texttt{a}, \texttt{C})$ to denote, respectively: the set of class names declared in P; the subtyping relation in P (which is always assumed to be acyclic); the class declaration CD of `C` in P when it exists; and the declaration of the attribute `a` in the closest superclass of `C` (including `C` itself) that contains a declaration for `a` in P, when it exists.

declarations $\overline{N}$ (e.g., classes, attributes, parameters, etc.) we assume that they have pairwise distinct names. Moreover, we write $names(\overline{N})$ to denote the sequence of the names of the declarations in $\overline{N}$.

For sake of readability, in the examples presented throughout the paper we use the Java syntax for operations on strings and sequential composition—encoding in IFJ syntax is straightforward (see [7] for examples). In the examples we also assume the existence of a built-in class `Int` for representing integer values.

Example 1 (A program for expressions) We illustrate the IFJ language with a simple program encoding the following grammar of numerical expressions:

```
Exp ::= Lit | Add
Lit ::= non-negative-integers
Add ::= Exp "+" Exp
```

The program, presented in Figure 4 (top), consists of: a class `Exp` representing all expressions; a class `Lit` representing literals; and a class `Add` representing an addition between two expressions.

All these classes implement a method `toInt` that computes the value of the expression, and a method `toString` that gives a textual representation of the expression. Note that class `Exp` is too general to provide a meaningful implementation of these methods. Indeed, it is supposed to be used only as a type and should never be instantiated.

3 Delta-Oriented SPLs

The IFΔJ language builds upon IFJ [7], adding to it a new layer for the implementation of the different DOP elements. The abstract syntax of IFΔJ SPLs is given in Figure 2—explanations are given in the caption. Recall that, according to Notation 1, we assume that the deltas declared in an artifact base have distinct names, the class operations in each delta act on distinct classes, the attribute operations in each class operation act on distinct attributes, etc.

In IFΔJ there is no concrete syntax for the FM and CK: it considers extensional representations. Namely, it represents feature models $\mathcal{M}$ by pairs "(set of features, set of products)" and configuration knowledge $\mathcal{K}$ by pairs "(delta activation map, delta application order)", see the following two definitions.

Definition 1 (Feature model) A *feature model* $\mathcal{M}$ is a pair $(\mathcal{F}, \mathcal{P})$ where $\mathcal{F}_x$ is a set of features and $\mathcal{P} \subseteq 2^{\mathcal{F}}$ is a set of products.

Definition 2 (Configuration knowledge for a delta-oriented SPL) CK for a delta-oriented SPL `L` is a pair $\mathcal{K}_{\texttt{L}} = (\alpha_{\texttt{L}}, <_{\texttt{L}})$ where: $\alpha_{\texttt{L}}$ is a map that associates to each delta name the set of products that activate it (the *delta activation map*); and $<_{\texttt{L}}$ is an ordering between delta names (the *delta application order*).

We have now all the ingredients for defining the notion of IFΔJ SPL.

LD ::= **line** L $\{\mathcal{M}\ \mathcal{K}\ AB\}$ — SPL Declaration
AB ::= $P\ \overline{DD}$ — Artifact Base
DD ::= **delta** d$\{\overline{CO}\}$ — Delta Declaration
CO ::= **adds** CD | **removes** C | **modifies** C[**extends** C′]$\{\overline{AO}\}$ — Class Operation
AO ::= **adds** AD | **removes** a | **modifies** MD — Attribute Operation

Fig. 2: **IFΔJ SPLs.** An SPL declaration comprises the name L of the product line, a feature model $\mathcal{M}$, a configuration knowledge $\mathcal{K}$, and an artifact base AB. The artifact base comprises a (possibly empty) IFJ program P, and a set of deltas $\overline{DD}$. A delta declaration DD comprises the name d of the delta and class operations $\overline{CO}$ representing the transformations performed when the delta is applied to an IFJ program. A class operation can add, remove, or modify a class. A class can be modified by (possibly) changing its super class and performing attribute operations $\overline{AO}$ on its body. An attribute operation can add or remove fields and methods, and modify the implementation of a method by replacing its body. The new body may call the special method name `original`, which is implicitly bound to the previous implementation of the method. FM, CK, AB (cf. the Introduction) of an SPL named L are denoted by $\mathcal{M}_{\mathtt{L}} = (\mathcal{F}_{\mathtt{L}}, \mathcal{P}_{\mathtt{L}})$, $\mathcal{K}_{\mathtt{L}} = (\alpha_{\mathtt{L}}, <_{\mathtt{L}})$ and $AB_{\mathtt{L}}$, respectively.

Definition 3 (Delta-oriented SPL) A delta-oriented SPL L is an SPL defined by means of the syntax in Figure 2.

The description of the generator of an IFΔJ SPL (given in Definition 4 below) relies on the two following auxiliary notions of applicable delta and application (of an applicable) delta.

- A delta d is *applicable* to a program P iff each class to be added does not already exist; each class to be removed or modified already exists; and for every class-modify operation: each method or field to be added does not yet exist; each method or field to be removed already exists; and each method to be modified already exists and has the same header specified in the method-modify operation.
- If a delta d is applicable to P, then the *application* of d to P is the program, denoted by $\mathtt{d}(P)$, obtained from P by applying all the operations in d—otherwise $\mathtt{d}(P)$ is undefined.

Definition 4 (Generator of a delta-oriented SPL, see [7]) The *generator* of L, denoted by $\mathcal{G}_{\mathtt{L}}$, is the mapping that associates each product p in $\mathcal{M}_{\mathtt{L}}$ with the IFJ program $\mathtt{d}_n(\cdots \mathtt{d}_1(P)\cdots)$, where P is the base program of L and $\mathtt{d}_1, \ldots, \mathtt{d}_n$ ($n \geq 0$) are the deltas of L activated by p (that are applied to P according to the application order).[1]

Note that the generator $\mathcal{G}_{\mathtt{L}}$ may be partial since, for some product of L, a delta $\mathtt{d}_i$ ($1 \leq i \leq n$) may not be applicable to the intermediate variant $\mathtt{d}_{i-1}(\cdots \mathtt{d}_1(P)\cdots)$ thus making $\mathcal{G}_{\mathtt{L}}$ undefined for that product.

Finally, since the rest of the paper focuses on the presentation of transformation algorithms for delta-oriented SPLs, we formalize the notion of equivalent SPLs, so we can prove that our refactoring algorithms are correct, i.e., they do not change the semantics of the input SPL.

Definition 5 (Delta-oriented SPL equivalence) Two delta-oriented SPLs are *equivalent* whenever they have the same feature model (see Definition 1) and generator (see Definition 4).

For sake of readability, in the examples presented throughout the paper, the feature model is represented by a feature diagram and the activation conditions of the deltas are expressed as propositional logic formulas ϕ where propositional variables are features f (i.e., the mapping $\alpha_{\mathtt{L}}$ is represented as a mapping from delta names d to propositional formulas ϕ). A formula ϕ represents the set of products

$$\{\overline{f} \mid \phi \text{ evaluates to } \textbf{true} \text{ when the variables } \overline{f} \text{ are } \textbf{true} \text{ and the other variables are } \textbf{false}\}$$

(see [4] for a discussion on other possible representations) and is described with the following syntax:

$$\phi ::= \textbf{true} \mid f \mid \phi \Rightarrow \phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \veebar \phi$$

where $\veebar$ is the xor operator.

Example 2 (The Expression Product Line) We illustrate the IFΔJ language with an example derived from the Expression Product Line (EPL) benchmark [41] (see also [44]), where the base program is given in Figure 4

[1] We assume *unambiguity* of the considered delta-oriented SPLs, i.e., for each product, any total order of the activated deltas that respects the (possibly partial) order specified in $<_{\mathtt{L}}$ generates the same variant—see [37, 7] for effective means to ensure unambiguity.

(top). Consider the following grammar of numerical expressions that extends the one of Example 1 with negation:

```
Exp ::= Lit | Add | Neg
Lit ::= non-negative-integers
Add ::= Exp "+" Exp
Neg ::= "-" Exp
```

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluating, which returns the value of the expression, either as an integer (`Int`) or as a literal expression (`Lit`).

Figure 3 shows the FM and CK of the EPL. Variability in the EPL can be described by two sets of features: the ones concerned with the data are Lit (for literals), Add (for the addition) and Neg (for the negation); the ones concerned with the operations are Print (for the classic `toString` method), Eval1 (for the `eval` method returning an int) and Eval2 (for the `eval` method returing a literal expression). The features Lit is mandatory, while Add, Neg, Print, Eval1 and Eval2 are optional. Moreover:

- as Eval1 and Eval2 define the same method, they are mutually exclusive, and
- at least one feature concerned with operations (i.e., either Print or one one among Eval1 and Eval2) must be selected.

The activation conditions of deltas in the CK mention only *concrete* features (i.e., the leaves in the feature diagram representing the FM), while the following propositional formula over concrete features provides an alternative specification of the FM: Lit $\wedge$ (Print $\vee$ (Eval1 $\veebar$ Eval2).

The artifact base of the EPL is given in Figure 4. The delta `DNeg` adds the class `Neg` with a simple setter. The delta `DNegPrint` adds to class `Neg` the `toString` method (relevant for the Print feature). The delta `DOptionalPrint` adds glue code to ensure that the two optional features Add and Neg cooperate properly: it *modifies* the implementation of the `toString` method of the class `Add` by putting parentheses around the textual representation of a sum expression, thus avoiding ambiguity in printing. This delta illustrates the usage of the special method `original` which allows here to call the original implementation of the method `toString`, and surround the resulting string with parentheses. The delta `DNegToint` adds to class `Neg` the `toInt` method (relevant for implementing the Eval feature).

The delta `DEval1` (resp. `DEval2`) modifies the class `Exp` by adding to them the `eval` method corresponding to the Eval1 (resp. Eval2) feature: `eval` takes no parameter and returns an `Int` (resp. a `Lit` object).

The delta `DremExpLitToint` removes the `toInt` method from the classes `Exp` and `Lit` when the feature `Eval` is not selected (more explicitly, when none of the features `Eval1` and `Eval2` is selected), and the delta `DremAddToint` does the same for the class `Add`.

Similarly, `DremExpLitPrint` removes the `toString` method from the classes `Exp` and `Lit` when the feature `Print` is not selected, and `DremAddToint` does the same for the class `Add`.

Finally, the delta `DremAdd` removes the class `Add` from the program when the feature `Add` is not selected.

4 Refactoring Delta-Oriented SPLs into Atomic Form

In this section, we present a first refactoring algorithm that simplifies the inner structure of deltas, in order to simplify as much as possible our main refactoring algorithms. More precisely, this algorithm refactors a delta-oriented SPL into *atomic* form, i.e., a normal form where each delta contains one operation.

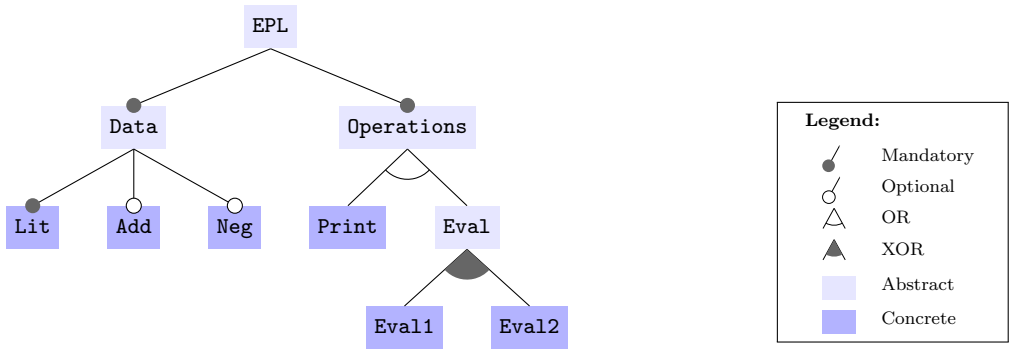
Definition 6 (Atomic SPL) An atomic SPL `L` (aSPL) is an SPL defined by means of the syntax in Figure 5.

The syntax in Figure 5 describes a subset of language described by the syntax in Figure 2. In particular, in the artifact base, the base program is empty and each delta contains a single operation. We remark that each attribute operation AO (cf. Figure 2) performs a single operation, namely AO can be either **adds** AD or **removes** `a` or **modifies** MD. Therefore, there are 6 possible atomic class operations: (empty) class-addition, class-remove, extend-modification (where $\texttt{C}'$ cannot be `Object`), attribute-addition, attribute-removal and method modification.

We start by devising an algorithm that refactors an IFΔJ SPL by replacing a given non-atomic delta by a set of atomic deltas.

Algorithm 1 (Atomic refactoring of a delta) Let `L` be an SPL containing a delta `d` with a body not atomic (in the sense of Definition 6). The following items describe how to generate an SPL $\texttt{L}^\star$ equivalent to `L`, where `d` has been replaced by a sequence of atomic deltas.

- $\texttt{L}^\star$ inherits the FM from `L` without modifications.
- $\texttt{L}^\star$ inherits the AB of `L`, where `d` is replaced by a sequence of atomic (freshly named) deltas $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ (where $n \geq 1$) that produce the same modifications of `d` whenever the list is applied in order. The decomposition of `d` in the list $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ is straightforward: (i) a class-remove operation is already atomic; (ii) an attribute operation is already atomic up to an



Activations:

Delta	Activation
DNeg	Neg
DNegPrint	Neg ∧ Print
DOptionalPrint	Add ∧ Neg ∧ Print
DNegToint	Neg ∧ (Eval1 ∨ Eval2)
DEval1	Eval1
DEval2	Eval2

Delta	Activation
DremExpLitToint	¬(Eval1 ∨ Eval2)
DremAddToint	Add ∧ ¬(Eval1 ∨ Eval2)
DremExpLitPrint	¬Print
DremAddPrint	Add ∧ ¬Print
DremAdd	¬Add

Order:

$\{\texttt{DNeg}\} <_{\texttt{L}} \{\texttt{DNegPrint}, \texttt{DOptionalPrint}, \texttt{DNegToint}, \texttt{DEval1}\}$
$<_{\texttt{L}} \{\texttt{DEval2}\}$
$<_{\texttt{L}} \{\texttt{DremExpLitToint}, \texttt{DremAddToint}, \texttt{DremExpLitPrint}, \texttt{DremAddPrint}\}$
$<_{\texttt{L}} \{\texttt{DremAdd}\}$

Fig. 3: Feature Model (top) and Configuration Knowledge (bottom) of the EPL

extend-modification that can be isolated in a suitable additional atomic operation; and (iii) a class addition can be atomized in term of empty-class creation, extend-tuning, field and method additions.
- $\texttt{L}^\star$ inherits the application condition of L, in the sense that for every delta d in L the activation conditions of $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ are set to the activation condition of d: we have $\alpha_{\texttt{L}^\star}(\texttt{d}_i^\star) = \alpha_{\texttt{L}}(\texttt{d})$ for all $1 \leq i \leq n$.
- $\texttt{L}^\star$ inherits the application order of L, where d is replaced by $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ totally ordered as listed. More precisely: if $\texttt{d}_0, \texttt{d}_1 \neq \texttt{d}$ are deltas in L, then (i) $\texttt{d}_0 <_{\texttt{L}^\star} \texttt{d}_1$ iff $\texttt{d}_0 <_{\texttt{L}} \texttt{d}_1$; (ii) $\texttt{d}_i^\star <_{\texttt{L}^\star} \texttt{d}_0$ iff $\texttt{d} <_{\texttt{L}} \texttt{d}_0$; (iii) $\texttt{d}_0 <_{\texttt{L}^\star} \texttt{d}_i^\star$ iff $\texttt{d}_0 <_{\texttt{L}} \texttt{d}$; and (iv) $\texttt{d}_i^\star <_{\texttt{L}^\star} \texttt{d}_j^\star$ iff $1 \leq i < j \leq n$. □

It is straightforward to check that the above procedure is effective and, in particular, the following result holds.

Lemma 1 (Correctness of the atomic refactoring of a delta) *Algorithm 1 describes a total transformation from an SPL* L *(with a total generator) into an equivalent (in the sense of Definition 5) SPL* $\texttt{L}^\star$ *containing less non-atomic deltas.*

Proof The termination of the algorithm is straightforward (i.e. the transformation is total). A generic delta d is linearly decomposed in an ordered list of atomic deltas $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ performing the same modifications. Since the ordered application of $\texttt{d}_1^\star, ..., \texttt{d}_n^\star$ produces the same modifications of d, the SPL $\texttt{L}^\star$ is by definition equivalent to L. □

The following algorithm transforms any IFΔJ SPL into an equivalent atomic SPL.

Algorithm 2 (Atomic refactoring of an SPL) Let **line** L $\{\mathcal{M}\ \mathcal{K}\ AB\}$ be an SPL such that $AB = P\ \overline{DD}$. The following steps describe how to generate an aSPL $\texttt{L}^\star$ equivalent to L.

First, we transform L into another (non-atomic SPL) $\texttt{L}'$ having an empty initial program as follows.

- $\texttt{L}'$ inherits the FM from L without modifications.
- $\texttt{L}'$ inherits the AB of L extended with a set of new (freshly named) deltas. For each class $\texttt{C}_i$ occurring in P, we include a class-addition delta $\texttt{d}_i^P$ that adds the class $\texttt{d}_i^P$ defined in P with all its fields and methods.
- $\texttt{L}'$ inherits the application condition of all deltas different from the new ones, by L. Moreover, the activation condition of $\texttt{d}_i^P$ are set to true.
- $\texttt{L}'$ inherits the application order of L modified to place before all new deltas to that coming from L. No order is imposed between the new deltas.

```
class Exp extends Object {
  Int toInt() { return null; }
  String toString() { return null; }
}

class Lit extends Exp {
  Int val;
  Lit setLit(Int x) { this.val=x; return this; }
  Int toInt() { return this.val; }
  String toString() { return this.val.toString(); }
}

class Add extends Exp {
  Exp a;
  Exp b;
  Add setAdd(Exp x, Exp y) { this.a=x; this.b=y; return this; }
  Int toInt() { return this.a.toInt().add(this.b.toInt()); }
  String toString() { return this.a.toString() + "+" + this.b.toString(); }
}
```

```
delta DNeg {
 adds class Neg extends Exp {
  Exp a;
  Neg setNeg(Exp x) { a = x; return this; }
}}

delta DNegPrint { modifies Neg { adds String toString() { return "−" + a.toString(); }}}

delta DOptionalPrint { modifies Add { modifies String toString() { return "(" + original() + ")"; }}}

delta DNegToint { modifies Neg { adds Int toInt() { return this.a.toInt().neg(); }}}

delta DEval1 { modifies Exp { adds Int eval() { return this.toInt(); }}}

delta DEval2 { modifies Exp { adds Lit eval() { return new Lit().setLit(this.toInt()); }}}
```

```
delta DremExpLitToint {
 modifies Exp { removes toInt; }
 modifies Lit { removes toInt; }
}

delta DremAddToint { modifies Add { removes toInt; }}

delta DremExpLitPrint {
 modifies Exp { removes toString; }
 modifies Lit { removes toString; }
}

delta DremAddPrint { modifies Add { removes toString; }}

delta DremAdd { removes Add }
```

Fig. 4: AB of the EPL: base program (top), deltas that add behaviour (middle) and deltas that remove behaviour (bottom)

$$
\begin{array}{rll}
\mathtt{a}LD & ::= & \textbf{line}\ \mathtt{L}\ \{\mathcal{M}\ \mathcal{K}\ \mathtt{a}AB\} \qquad \text{Atomic SPL Declaration} \\
\mathtt{a}AB & ::= & \overline{\mathtt{a}DD} \qquad \text{Atomic Artifact Base} \\
\mathtt{a}DD & ::= & \textbf{delta}\ \mathtt{d}\{\mathtt{a}CO\} \qquad \text{Atomic Delta Declaration} \\
\mathtt{a}CO & ::= & \textbf{adds class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{Object}\ \{\ \} \mid \textbf{removes}\ \mathtt{C} \\
 & \mid & \textbf{modifies}\ \mathtt{C}\ \textbf{extends}\ \mathtt{C}'\ \{\ \} \mid \textbf{modifies}\ \mathtt{C}\ \{AO\} \qquad \text{Atomic Class Operation}
\end{array}
$$

Fig. 5: **Atomic SPL.** Above, we assume that $\mathtt{C}'$ cannot be `Object`. For sake of conciseness, we shorten the class-addition and the extend-modification in **adds class** `C` **extends** `Object` and **modifies** `C` **extends** $\mathtt{C}'$.

Finally, we transform each non-atomic delta of $\mathtt{L}'$ into a set of atomic deltas by repeatedly applying Algorithm 1 until all non-atomic deltas have been eliminated. □

It is straightforward to see that the above procedure is effective and, in particular, the following theorem holds.

Theorem 1 (Correctness of the atomic refactoring of an SPL) *Algorithm 2 describes a total transformation from an SPL (with a total generator) into an equivalent (in the sense of Definition 5) atomic SPL.*

Proof The termination of the algorithm is immediate: it first transforms each class in the initial program into a delta (this step finishes as the initial program has a finite number of classes); and then applies Algorithm 1 on every delta of the resulting product line (this step finishes as there is only a finite number of deltas). Patently, the produced SPL is atomic. Namely, its initial program is empty (it has been translated into deltas) and each of its deltas is atomic (i.e., it follow the syntax given in Figure 5). □

Example 3 (Atomic refactoring of the EPL) Figures 6 and 7 present the CK and AB of the SPL produced by the application of Algorithm 2 on the EPL presented in Example 2. The different classes of the base program of the EPL have been translated into a set of atomic deltas. For instance, the deltas whose names start with `DLit` construct the `Lit` class by: (i) creating the class (with the delta `DLit`); (ii) making it extend `Exp` (with the delta `DLitExtends`); (iii) creating its field `val` (with the delta `DLitVal`); and (iv) creating its different methods (with the deltas `DLitSetlit`, `DLitToint` and `DLitTostring`). Similarly, a set of deltas with related names construct the `Exp` and `Add` classes.

Additionally, in the original EPL, the deltas `DDNeg`, `DremExpLitToint` and `DremExpLitPrint` were not atomic. Applying the Algorithm 2 on the EPL thus splits each of these deltas into atomic deltas.

5 Refactoring Atomic Delta-Oriented SPLs into Remove-free and Replace-free Forms

The two refactoring algorithms use the following auxiliary definition.

Definition 7 (Principal ideal of a delta) Let `L` be an SPL and `d` be the name of a delta in it. The *principal ideal* at `d` w.r.t. the application order of `L`, written $\downarrow^{\mathtt{L}}\mathtt{d}$, is the set $\{\mathtt{d}' \mid \mathtt{d}' \leq_{\mathtt{L}} \mathtt{d}\}$.

By exploiting the above notation we can reformulate the unambiguity assumption (cf. the footnote at the end of Definition 4) with a local flavour: an SPL `L` is unambiguous whenever any total order $\leq_t$ of the activated deltas that respects the principal ideal of all activated `d` (namely, each activated delta precedes the deltas in $\downarrow^{\mathtt{L}}\mathtt{d}$) generates the same variant.

5.1 Refactoring into Remove-free Form

This refactoring consists in the transformation of an atomic SPL into a remove-free one, i.e., an SPL where deltas of the form **removes** `C` and **modifies** `C` {**removes** `a`} have been eliminated. The following definition formalizes the notion of remove-free SPL.

Definition 8 (Remove-free aSPL) An aSPL is *remove-free* iff it does not contain remove operations. More precisely, it is defined from the syntax in Figure 8.

The syntax in Figure 8 is a restriction of the syntax provided in Figure 5. We eliminated both class-remove and attribute-remove operations; then, we wrote explicitly the possible remaining attribute operations.

We structure our refactoring algorithm in three parts: the first part eliminates a remove-class operation, the second eliminates a remove-attribute operation, and the third one combines both into the full refactoring algorithm. The algorithm responsible for the elimination of a remove-class operation is the following.

Algorithm 3 (Remove-class operation elimination refactoring) Let `L` be an aSPL containing a delta `d` defined as follows for some class name `C`:

Activations:

Delta	Activation
DExp	True
DExpToint	True
DExpTostring	True
DLit	True
DLitExtends	True
DLitVal	True
DLitSetlit	True
DLitToint	True
DLitTostring	True
DAdd	True
DAddExtends	True
DAddA	True
DAddB	True
DAddSetadd	True
DAddToint	True
DAddTostring	True

Delta	Activation
DNeg	Neg
DNegExends	Neg
DNegA	Neg
DNegSetneg	Neg
DNegPrint	Neg ∧ Print
DOptionalPrint	Add ∧ Neg ∧ Print
DNegToint	Neg ∧ (Eval1 ∨ Eval2)
DEval1	Eval1
DEval2	Eval2

Delta	Activation
DremExpToint	¬(Eval1 ∨ Eval2)
DremLitToint	¬(Eval1 ∨ Eval2)
DremAddToint	Add ∧ ¬(Eval1 ∨ Eval2)
DremExpPrint	¬Print
DremLitPrint	¬Print
DremAddPrint	Add ∧ ¬Print
DremAdd	¬Add

Order:

{DExp, DLit, DAdd}
$<_L$ {DExpToint, DExpTostring,
DLitExtends, DLitVal, DLitSetlit, DLitToint, DLitTostring,
DAddExtends, DAddA, DAddB, DAddSetadd, DAddToint, DAddTostring}
$<_L$ {DNeg}
$<_L$ {DNegExtends, DNegA, DNegSetneg}
$<_L$ {DNegPrint, DOptionalPrint, DNegToint, DEval1}
$<_L$ {DEval2}
$<_L$ {DremExpToint, DremLitToint, DremAddToint, DremExpPrint, DremLitPrint, DremAddPrint}
$<_L$ {DremAdd}

Fig. 6: CK of the atomic version of the EPL

delta d { **removes** C }.

The following items describe how to generate an aSPL $\mathtt{L}^\star$ equivalent to L, where d has been eliminated.

- $\mathtt{L}^\star$ inherits the FM from L without modifications.
- $\mathtt{L}^\star$ inherits the AB of L, where d is removed.
- $\mathtt{L}^\star$ inherits the application order of L, where d is removed.
- $\mathtt{L}^\star$ inherits the application condition of L for all deltas, except for the ones in $\downarrow^{\mathtt{L}}\mathtt{d}$ that operate on the class C (more precisely, deltas having one of the following four shapes: **adds class** C **extends** Object, **modifies** C **extends** C′, **modifies** C {**adds** *AD*} and **modifies** C {**modifies** *MD*}). If d′ is one such delta then its activation condition in $\mathtt{L}^\star$ is set to $\alpha_{\mathtt{L}}(\mathtt{d}') \setminus \alpha_{\mathtt{L}}(\mathtt{d})$.[2] □

It is clear that the above procedure is effective and, in particular, the following result holds.

[2] If the activation conditions of deltas are expressed by propositional formulas over features, then the mapping $\alpha_{\mathtt{L}}$ is represented as a mapping from delta names d to propositional formulas ϕ (cf. the explanation before Example 2) and the activation condition is set to $\alpha_{\mathtt{L}}(\mathtt{d}') \wedge \neg\alpha_{\mathtt{L}}(\mathtt{d})$.

Lemma 2 (Correctness of the remove-class operation elimination refactoring) *Algorithm 3 describes a transformation from an aSPL (inducing a total generator) into an equivalent aSPL (inducing a total generator).*

Proof The termination of the algorithm is immediate and the produced SPL is atomic. Let D be the set of deltas in $\downarrow^{\mathtt{L}}\mathtt{d}$ that operate on the class C, we note that $\alpha_{\mathtt{L}}(\mathtt{d}) \subseteq \cup_{\mathtt{d}' \in D}\alpha_{\mathtt{L}}(\mathtt{d}')$ because the totality of the generator (cf. Definition 4) ensures that the class is added by a delta before being removed, for each product in $\alpha_{\mathtt{L}}(\mathtt{d})$. The modifications of the activation conditions of deltas in D have two main effects: to avoid useless modification of a class when it has to be removed and to ensure that the generator is still well defined, because deltas are still applied only in applicable cases (cf. notion defined before Definition 4). □

The following algorithm describes the elimination of an attribute-remove operation.

Algorithm 4 (Remove-attribute operation elimination refactoring) Let L be an aSPL containing a delta d defined as follows for some attribute a and class C:

```
delta DExp { adds class Exp extends Object { }}
delta DExpToint { modifies Exp { adds Int toInt() { return null; }}}
delta DExpTostring { modifies Exp { adds String toString() { return null; }}}

delta DLit { adds class Lit extends Object { }}
delta DLitExtends { modifies Lit extends Exp { }}
delta DLitVal { modifies Lit { adds int val; }}
delta DLitSetlit { modifies Lit { adds Lit setLit(Int x) { this.val=x; return this; }}}
delta DLitToint { modifies Lit { adds Int toInt() { return this.val; }}}
delta DLitTostring { modifies Lit { adds String toString() { return this.val.toString(); }}}

delta DAdd { adds class Add extends Object { }}
delta DAddExtends { modifies Add extends Exp { }}
delta DAddA { modifies Add { adds Exp a; }}
delta DAddB { modifies Add { adds Exp b; }}
delta DAddSetadd { modifies Add { adds Add setAdd(Exp x, Exp y) { this.a=x; this.b=y; return this; }}}
delta DAddToint { modifies Add { adds Int toInt() { return this.a.toInt().add(this.b.toInt()); }}}
delta DAddTostring { modifies Add {
 adds String toString() { return this.a.toString() + "+" + this.b.toString(); }}}
```

```
delta DNeg { adds class Neg extends Object { }}
delta DNegExtends { modifies Neg extends Exp { }}
delta DNegA { modifies Neg { adds Exp a; }}
delta DNegSetneg { modifies Neg { adds Neg setNeg(Exp x) { a = x; return this; }}}

delta DNegPrint { modifies Neg { adds String toString() { return "−" + a.toString(); }}}

delta DOptionalPrint { modifies Add { modifies String toString() { return "(" + original() + ")"; }}}}

delta DNegToint { modifies Neg { adds Int toInt() { return this.a.toInt().neg(); }}}}

delta DEval1 { modifies Exp { adds Int eval() { return this.toInt(); }}}}

delta DEval2 { modifies Exp { adds Lit eval() { return new Lit().setLit(this.toInt()); }}}
```

```
delta DremExpToint { modifies Exp { removes toInt; }}
delta DremLitToint { modifies Lit { removes toInt; }}

delta DremAddToint { modifies Add { removes toInt;}}

delta DremExpPrint { modifies Exp { removes toString; }}
delta DremLitPrint { modifies Lit { removes toString; }}

delta DremAddPrint { modifies Add { removes toString; }}

delta DremAdd { removes Add }
```

Fig. 7: AB of the atomic version of the EPL: deltas from the base program (top); deltas that add behaviour (middle); and deltas that remove behaviour (bottom)—the deltas highlighted in grey are the same as in Figure 4

delta `d` { **modifies** `C` { **removes** `a` } }.

The following items describe how to generate an aSPL $\mathtt{L}^\star$ equivalent to `L`, where $\mathtt{d}^*$ has been eliminated.

- $\mathtt{L}^\star$ inherits the FM from `L` without modifications.
- $\mathtt{L}^\star$ inherits the AB of `L`, where `d` is removed.
- $\mathtt{L}^\star$ inherits the application order of `L`, where `d` is removed.
- For the application conditions we consider two subcases.

$\mathtt{a}LD$ ::= **line** L $\{\mathcal{M}\ \mathcal{K}\ \mathtt{a}AB\}$ Atomic SPL Declaration
$\mathtt{a}AB$ ::= $\overline{\mathtt{a}DD}$ Atomic Artifact Base
$\mathtt{a}DD$ ::= **delta** d$\{\mathtt{a}CO\}$ Atomic Delta Declaration
$\mathtt{a}CO$ ::= **adds class** C **extends** Object | **modifies** C **extends** C′
| **modifies** C {**adds** AD} | **modifies** C {**modifies** MD} Atomic Operation

Fig. 8: **Remove-Free SPL.** In this syntax, C′ cannot be Object.

1. Case where a is a field. Let $\mathtt{d}_1, \ldots, \mathtt{d}_n$ (where $n \geq 0$) be the deltas of the shape

   modifies C {**adds** AD}

   that occur in $\downarrow^{\mathtt{L}}\mathtt{d}$, where the name of AD is a. We set the activation condition of these deltas in $\mathtt{L}^\star$ as follows: $\alpha_{\mathtt{L}^\star}(\mathtt{d}_i) = \alpha_{\mathtt{L}}(\mathtt{d}_i) \setminus \alpha_{\mathtt{L}}(\mathtt{d})$ for all $1 \leq i \leq n$. All other deltas in $\mathtt{L}^\star$ inherit the activation conditions from L.
2. Case where a is a method. Let $\mathtt{d}_1, \ldots, \mathtt{d}_n$ (where $n \geq 0$) be the deltas of the shape

   modifies C {**adds** MD}

   that occur in $\downarrow^{\mathtt{L}}\mathtt{d}$, where the name of MD is a. Moreover, let $\mathtt{d}_{n+1}, \ldots, \mathtt{d}_{n+m}$ (for some $m \geq 0$) be the deltas of the shape

   modifies C {**modifies** MD}

   that occur in $\downarrow^{\mathtt{L}}\mathtt{d}^*$, where the name of MD is a. We set activation condition of these deltas in $\mathtt{L}^\star$ as follows: $\alpha_{\mathtt{L}^\star}(\mathtt{d}_i) = \alpha_{\mathtt{L}}(\mathtt{d}_i) \setminus \alpha_{\mathtt{L}}(\mathtt{d})$ for all $1 \leq i \leq n+m$. All other deltas in $\mathtt{L}^\star$ inherit the activation conditions from L. □

The above procedure is effective and, in particular, the following result holds.

Lemma 3 (Correctness of the remove-attribute operation elimination refactoring) *Algorithm 4 describes a transformation from an aSPL (inducing a total generator) into an equivalent aSPL (inducing a total generator).*

Proof The termination of the algorithm is immediate and the resulting SPL is atomic. The deltas $\mathtt{d}_1, \ldots, \mathtt{d}_n$ ($n \geq 0$) of the shape **modifies** C {**adds** a} that occur in $\downarrow^{\mathtt{L}}\mathtt{d}$ are the ones that can play some role in the applicability of **modifies** C {**removes** a}. In particular, since the generator is total it must happen that $\cup_{i=1}^{n}\alpha_{\mathtt{L}}(\mathtt{d}_i) \subseteq \alpha_{\mathtt{L}}(\mathtt{d})$, viz. a is certainly added before being modified by d. If a is a method then, after the addition, it can be (uselessly) modified by a delta of the shape **modifies** C {**modifies** MD} that occurs in $\downarrow^{\mathtt{L}}\mathtt{d}$. The restriction of the activation conditions of the considered deltas in $\downarrow^{\mathtt{L}}\mathtt{d}$ avoids the activation of additions/modifications of attributes, in all cases in which a is removed. □

We can now present the following algorithm, that transforms any IFΔJ aSPL into an equivalent remove-free aSPL.

Algorithm 5 (Remove-free refactoring of an aSPL) Let L be an aSPL. In order to generate a remove-free aSPL $\mathtt{L}^\star$ equivalent to L, apply repeatedly the Algorithms 3 and 4 until all remove-operations have been eliminated from L. □

Theorem 2 (Correctness of the remove-free refactoring of an aSPL) *Algorithm 5 describes a transformation from an aSPL (inducing a total generator) into an equivalent remove-free aSPL (inducing a total generator).*

Proof This result is a direct consequence of Lemma 2 and Lemma 3. □

It is worth observing that Algorithm 5 eliminates all the deltas comprising a remove operation, and only changes the activation condition of some of the remaining deltas.

Example 4 (Remove-free refactoring of the atomic EPL) The application of Algorithm 5 on the atomic version of the EPL presented in Example 3 removes the deltas in the bottom part of Figure 7 and modifies the application conditions of some of the deltas as illustrated in Figure 9.

5.2 Refactoring into Replace-free Form

We now discuss the refactoring algorithm that removes the *modifies* operation that does not call original. We call the SPL without such delta operation *replace-free* SPL, as formalized in the following definition.

Definition 9 (Replace-free aSPL) An aSPL is *replace-free* whenever it is remove-free (see Definition 8) and it does not contain any method-modifications operations that do not call original (cf. caption of Figure 2).

Activations:

Delta	Activation
DExp	True
DExpToint	Eval1 ∨ Eval2
DExpTostring	Print
DLit	True
DLitExtends	True
DLitVal	True
DLitSetlit	True
DLitToint	Eval1 ∨ Eval2
DLitTostring	Print
DAdd	Add
DAddExtends	Add
DAddA	Add
DAddB	Add
DAddSetadd	Add
DAddToint	Add ∧ (Eval1 ∨ Eval2)
DAddTostring	Add ∧ Print

Delta	Activation
DNeg	Neg
DNegExends	Neg
DNegA	Neg
DNegSetneg	Neg
DNegPrint	Neg ∧ Print
DOptionalPrint	Add ∧ Neg ∧ Print
DNegToint	Neg ∧ (Eval1 ∨ Eval2)
DEval1	Eval1
DEval2	Eval2

Fig. 9: Activation conditions for the deltas of the remove-free version of the EPL—the activation conditions highlighted in grey are the same as in Figure 6, in particular: the activation condition of the delta `DOptionalPrint` (written in red) has been processed by the algorithm and the produced activation condition is equivalent to the original one, while all the other activation conditions highlighted in grey have not been processed by the algorithm

Like previously, we structure our refactoring algorithm in two parts: the first one changes a delta containing a method-replace operation into a delta containing a method-add operation, and the second one applies the first one on all such deltas, thus changing them all.

Algorithm 6 (Replace-method operation elimination refactoring) Let $\mathtt{L}$ be a remove-free aSPL containing a delta $\mathtt{d}$ defined as follows for some class $\mathtt{C}$ and some method *MD* named $\mathtt{m}$ that does not call `original`:

delta $\mathtt{d}${**modifies** $\mathtt{C}$ {**modifies** *MD*}}

The following items describe how to generate an aSPL $\mathtt{L}^\star$ equivalent to $\mathtt{L}$, where $\mathtt{d}$ has been replaced with a new delta $\mathtt{d}'$ containing the operation **modifies** $\mathtt{C}$ {**adds** *MD*}.

- $\mathtt{L}^\star$ inherits the FM from $\mathtt{L}$ without modifications.
- $\mathtt{L}^\star$ inherits the AB of $\mathtt{L}$, where $\mathtt{d}$ is replaced by $\mathtt{d}'$.
- $\mathtt{L}^\star$ inherits the application order of $\mathtt{L}$, where $\mathtt{d}$ is replaced by $\mathtt{d}'$. More precisely, if $\mathtt{d}_0, \mathtt{d}_1 \neq \mathtt{d}, \mathtt{d}'$ then: (i) $\mathtt{d}_0 <_{\mathtt{L}^\star} \mathtt{d}_1$ iff $\mathtt{d}_0 <_{\mathtt{L}} \mathtt{d}_1$; (ii) $\mathtt{d}' <_{\mathtt{L}^\star} \mathtt{d}_1$ iff $\mathtt{d} <_{\mathtt{L}} \mathtt{d}_1$; and (iii) $\mathtt{d}_0 <_{\mathtt{L}^\star} \mathtt{d}'$ iff $\mathtt{d}_0 <_{\mathtt{L}} \mathtt{d}$.
- Let $\mathtt{d}_1, \ldots, \mathtt{d}_n$ (where $n \geq 0$) be all deltas of the shape **modifies** $\mathtt{C}$ {**adds** *MD*′} that occur in $\downarrow^{\mathtt{L}}\mathtt{d}^*$ and add a method named $\mathtt{m}$. Let $\mathtt{d}_{n+1}, \ldots, \mathtt{d}_{n+m}$ (where $m \geq 0$) be all deltas of the shape

  modifies $\mathtt{C}$ {**modifies** *MD*′}

  that occur in $\downarrow^{\mathtt{L}}\mathtt{d}^*$ and redefine $\mathtt{m}$. We set the activation condition of these deltas in $\mathtt{L}^\star$ as follows: $\alpha_{\mathtt{L}^\star}(\mathtt{d}_i) = \alpha_{\mathtt{L}}(\mathtt{d}_i) \setminus \alpha_{\mathtt{L}}(\mathtt{d})$ for all $1 \leq i \leq n+m$. All other deltas $\mathtt{d}''$ in $\mathtt{L}^\star$ inherit the activation conditions from $\mathtt{L}$, i.e. $\alpha_{\mathtt{L}^\star}(\mathtt{d}'') = \alpha_{\mathtt{L}}(\mathtt{d}'')$. □

The above procedure is effective and, in particular, the following lemma holds.

Lemma 4 (Correctness of the replace-method operation elimination refactoring) *Algorithm 6 describes a transformation from a remove-free aSPL (inducing a total generator) into an equivalent remove-free aSPL in which a method-replace operation has been replaced by a method-add operation.*

Proof It is straightforward to see that the algorithm terminates and that the resulting SPL is atomic. The deltas $\mathtt{d}_1, \ldots, \mathtt{d}_n$ $(n \geq 0)$ of the shape

modifies $\mathtt{C}$ {**adds** *MD*′}

that occur in $\downarrow^{\mathtt{L}}\mathtt{d}$ are the ones that can play some role in the applicability of **modifies** $\mathtt{C}$ {**modifies** *MD*}. In particular, since the generator is total it must happen that $\alpha_{\mathtt{L}}(\mathtt{d}) \subseteq \cup_{i=1}^{n} \alpha_{\mathtt{L}}(\mathtt{d}_i)$, viz. a method is certainly added before to be modified. Moreover, the method after the addition can be modified by a delta of the shape **modifies** $\mathtt{C}$ {**modifies** *MD*′} that occurs in $\downarrow^{\mathtt{L}}\mathtt{d}$. The restriction of the activation conditions of the considered deltas in $\downarrow^{\mathtt{L}}\mathtt{d}$ avoids the activation of additions/modifications of the method in all cases where the method was replaced. □

We can now present the following algorithm, that transforms any IFΔJ remove-free aSPL into an equivalent replace-free aSPL

Algorithm 7 (Replace-free refactoring of a remove-free aSPL) Let L be an aSPL. In order to generate a monotone-free aSPL $\mathtt{L}^\star$ equivalent to L, eliminate all operations **modifies** C {**modifies** *MD*} not involving `original` by repeatedly applying Algorithm 6. □

It is worth observing that Algorithm 7 transforms each of the deltas comprising a replace-method operation into a delta comprising an add-method operation, and only changes the activation condition of some of the remaining deltas.

Theorem 3 (Correctness of the replace-free refactoring of a remove-free aSPL) *Algorithm 7 describes a transformation from a remove-free aSPL (inducing a total generator) into an equivalent replace-free aSPL.*

Proof The proof follows by Lemma 4. □

Example 5 (Application of Algorithm 7) Consider a version of the atomic remove-free EPL of Example 4, where the delta `DOptionalPrint` has been changed as follows:

```
delta DOptionalPrint {
 modifies Add {
  modifies String toString() {
    return "(" + this.a.toString() + "+"
               + this.b.toString() + ")";
}}}
```

This new version of the delta `DOptionalPrint` does not call `original` and thus the refactoring Algorithm 7 transforms its modify-method operation into an add-method operation. Additionally, the activation condition of the delta `DAddTostring` is changed in order not to be in conflict with the new method-add operation. Figure 10 shows the definition of the delta `DOptionalPrint` and the activation condition of the delta `DAddTostring` after the application of Algorithm 7.

6 Discussion

The overall behaviour of the refactoring algorithms presented in this paper can be summarized as follows.

- Algorithm 2 splits each delta in atomic deltas. It does not change atomic deltas. Therefore, it behaves like the identity when applied to an SPL that is already in atomic form. It may increase the size of the CK and of the AB by a small constant factor. The correspondence between the original and the refactored SPLs is straightforward and suitable tool support can track it and provide to the developers of the SPL the two views.
- Algorithm 5 eliminates remove operations from an aSPL. It does not change remove-free deltas. Therefore, it behaves like the identity when applied to an SPL that is already in remove-free form. It does not increase the size of the AB. Clearly, this refactoring improves the comprehensibility of the aSPL.
- Algorithm 7 transforms a remove-free aSPL by transforming each replace-method operation into an add-method operation. It does not change replace-free deltas. Therefore, it behaves like the identity when applied to an SPL that is already in replace-free form. It does not increase the size of the AB. Also this refactoring improves the comprehensibility of the SPL, since the semantics of a modify-method operation that does not call `original` is conceptually more similar to that of a method-add operation.

These refactoring algorithms transform an SPL without requiring interaction with the developers of the SPL. However, in practice, SPL developers should do a final revision pass on the refactored SPL to improve its comprehensibility. For instance, to rename some deltas, to merge some deltas that have the same activation condition or to reintroduce a non-empty base program—an accordingly revised version of CK and AB of the remove-free EPL of Example 4 is illustrated in Figure 11 and Figure 12, respectively (cf. the original version of CK and AB the EPL of Example 2 in Figure 3 and Figure 4, respectively). In order to assist SPL developers during this final revision pass, it will be useful to develop suitable tool support that tracks the connection between the CK/AB of the refactored SPL and the CK/AB of the original SPL.

Both the remove-free form and the replace-free form could facilitate performing further simplifications that reduce the size of the AB like, e.g., detecting and merging equivalent deltas with different names.

7 Related Work

To the best of our knowledge, refactoring of delta-oriented SPLs has been studied only in the works by Schulze et. al [46], by Haber et al. [23], and in our previous work [17]. Schulze et. al present a catalogue of refactoring algorithms and code smells for delta-oriented SPLs of Java programs [34], while Haber et al. consider similar refactoring primitives for delta-oriented SPLs of software architectures. Most of the refactorings presented in [46] are based on object-oriented refactorings [22]. Two of their refactorings are related to ours: *Resolve Modification Action* replaces a **modifies** operations that does not call `original` with an **adds** operation, by modifying the activation condition of previous

```
delta DOptionalPrint {
 modifies Add {
  adds String toString() {
   return "(" + this.a.toString() + "+" + this.b.toString() + ")";
}}}
```

Activations:

Delta Module	Activation
`DAddTostring`	Add ∧ ¬Neg

Fig. 10: Delta `DOptionalPrint` (top) and activation condition of the delta `DAddTostring` (bottom) in the replace-free version of the EPL

Activations:

Delta	Activation
`DExpLitToint`	Eval1 ∨ Eval2
`DExpLitTostring`	Print
`DAdd`	Add
`DAddToint`	Add ∧ (Eval1 ∨ Eval2)
`DAddTostring`	Add ∧ Print

Delta	Activation
`DNeg`	Neg
`DNegPrint`	Neg ∧ Print
`DOptionalPrint`	Add ∧ Neg ∧ Print
`DNegToint`	Neg ∧ (Eval1 ∨ Eval2)
`DEval1`	Eval1
`DEval2`	Eval2

Order:

{`DAdd`}
$<_L$ {`DExpLitToint`, `DExpLitTostring`, `DAddToint`, `DAddTostring`, `DNeg`}
$<_L$ {`DNegPrint`, `DOptionalPrint`, `DNegToint`, `DEval1`}
$<_L$ {`DEval2`}

Fig. 11: CK of the revised refactored EPL

modifies and **adds** operations; and *Resolve Removal Action* eliminates **removes** operations also by changing the application condition of previous **modifies** and **adds** operations. However, the refactoring algorithms proposed in this paper perform an overall transformation on the whole SPL. In our previous work [17] we proposed algorithms to refactor any delta-oriented SPL into an equivalent one that follows guidelines that make type checking more efficient [15].

The *Feature-Oriented Programming* (FOP) [5] [3, Sect. 6.1] SPL implementation approach can be described as a restriction of DOP where deltas are associated one-to-one with features and have limited expressive power: they can add and modify program elements, however, they cannot remove them (see, e.g., [45] for a detailed comparison between DOP and FOP). Refactoring algorithms for FOP have been proposed in [2,39]. They focus on decomposing existing programs into features in order to support extractive SPL development [36]. Instead, our refactoring algorithms (like those proposed in [46] and [23]) focus on improving the structure of existing delta-oriented SPLs. The variant-preserving refactorings for FOP proposed by Schulze et al. [47] are essentially a subset of the refactoring of DOP proposed in [46].

Monteiro and Fernandes [42] presented a catalogue of refactorings for aspect-oriented programming written in AspectJ [33]. This work does not focus on SPLs and does not take variability into account. Instead, Kästner and Kuhlemann [32] propose a tool that supports refactoring legacy Java applications into features and generates an SPL implemented in the Jak language for FOP [5] or AspectJ.

Borba et al. [8] present a language-independent theory of SPL refinement for justifying stepwise and compositional product line evolution. This work has the same aim as our work, that is, supporting evolution and refactorings of SPLs. However, it focuses on feature model and configuration knowledge, while our work focuses on configuration knowledge and artifact base.

8 Conclusion and Future Work

In this paper, we introduced refactoring algorithms that aim at improving the comprehensibility of delta-oriented SPLs of Java-like programs. We have presented the refactoring algorithms for the *Imperative Featherweight*

```
class Exp extends Object { }

class Lit extends Exp {
  Int val;
  Lit setLit(Int x) { this.val=x; return this; }
}
```

```
delta DExpLitToint {
 modifies Exp { adds Int toInt() { return null; }
 modifies Lit { adds Int toInt() { return this.val; }
}}

delta DExpLitTostring {
 modifies Exp { adds String toString() { return null; }
 modifies Lit { adds String toString() { return this.val.toString(); }
 }}

 class Add extends Exp {
  Exp a;
  Exp b;
  Add setAdd(Exp x, Exp y) { this.a=x; this.b=y; return this; }
}

delta DAddToInt { modifies Add { adds Int toInt() { return this.a.toInt().add(this.b.toInt()); }}}
delta DAddToString { modifies Add {
 adds String toString() { return this.a.toString() + "+" + this.b.toString(); }}}
```

```
delta DNeg {
 adds class Neg extends Exp {
  Exp a;
  Neg setNeg(Exp x) { a = x; return this; }
}}

delta DNegPrint { modifies Neg { adds String toString() { return "−" + a.toString(); }}}

delta DOptionalPrint { modifies Add { modifies String toString() { return "(" + original() + ")"; }}}}

delta DNegToint { modifies Neg { adds Int toInt() { return this.a.toInt().neg(); }}}}

delta DEval1 { modifies Exp { adds Int eval() { return this.toInt(); }}}}

delta DEval2 { modifies Exp { adds Lit eval() { return new Lit().setLit(this.toInt()); }}}
```

Fig. 12: AB of the revised refactored EPL: base program (top); deltas from the base program of the original EPL (middle); and deltas that add behaviour (bottom)

Delta Java (IFΔJ) core calculus for delta-oriented SPLs. In IFΔJ there is no concrete syntax for FM and CK: it considers extensional representations. In future work we plan to specialize the proposed algorithms by considering concrete representations for FM and CK and to evaluate their computational complexity.

The toolchain of the HyVar project [10,38] supports the development of delta-oriented SPLs where the variants are statecharts [25] expressed in the format supported by Yakindu Statechart Tools [28]. In particular, delta-oriented SPLs of statecharts have been formalized by means of the core textual languages FSL (that capture the key ingredients of Yakindu statecharts) and FΔSL (for delta-oriented SPLs of FSL statcharts) [38]. In future work we would like to adapt the refactoring algorithms presented in this paper to FΔSL SPLs and to integrate them into the HyVar toolchain. Recently, Wille et al. [48] proposed a variability mining procedure that, given a set S of models (written in a given modeling language, e.g., statecharts) generated by clone-and-own industrial practice [20], semi-automatically identifies variability information (i.e., com-

mon and varying parts) on the elements of S, and then extracts from S a delta-oriented SPL of models. The procedure, which can be applied to different modeling languages, generates a delta language specifically tailored to transforming models in the analyzed modeling language. The procedure is evaluated by two case studies with industrial background that consider a set of *MATLAB/Simulink* models and a set of *Rational Rhapsody* statechart models, respectively. In future work we would like to adapt this variability mining procedure to extract delta-oriented SPLs expressed in the language of the HyVar toolchain and to evaluate whether applying the refactoring algorithms presented in this paper to the extracted SPLs produces some benefit.

FineFit [21] is an approach for model-based testing of Java programs which relies on the notion of *data refinement* [43] to compare the state of the model with the state of the *system under test* (SUT). DeltaFineFit [13] is a recently proposed model-based testing approach for delta-oriented SPLs written in DeltaJ [34,49] (a prototypical language for delta-oriented programming of SPLs of Java programs). DeltaFineFit integrates data-refinement-based testing into delta-oriented SPL development by ensuring that each product is generated together with its FineFit model, thus enabling the fully automated testing of all the products of an SPL.[3] In future work we would like to explore whether applying the DeltaFineFit approach to SPLs that are in remove- or replace-free form could result in the generation of tests that are more efficient to execute.

The Abstract Behavioural Specification (ABS) language [11] is a delta-oriented modeling language that has been successfully used in industry [31,26,1,14]. In future work we would like to formulate our refactoring algorithms for ABS and to implement them as part of of the ABS toolchain (`http://abs-models.org/`) and to apply them on concrete industrial case studies, in other to evaluate whether they allow to improve the comprehensibility of the considered SPLs.

Dynamic software product lines [24,9,6] address engineering adaptive systems by using a dedicated variability model describing all possible configurations a system may adapt to at runtime. Dynamic DOP [19] extends DOP with the capability to switch the implemented product configuration at runtime. A dynamic delta-oriented SPL is a delta-oriented SPL with a dynamic reconfiguration graph that specifies how to switch between different feature configurations. Dynamic DOP has been formalized by means of a core calculus that extends IFΔJ [18] and we are planning to implement dynamic DOP for ABS. In future work it would be interesting to evaluate whether considering SPLs that are in remove- or replace-free form could improve the efficiency of dynamic reconfiguration.

[3] When the number of products is too large, testing all the products is unfeasible. This could be addressed by using, e.g., *sample-based SPL testing* techniques [30,29,40,35], where a subset of products—covering relevant combinations of features—is generated and tested by applying single system testing techniques.

Acknowledgements We thank the anonymous reviewers for comments and suggestions for improving the presentation.

References

1. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. Service Oriented Computing and Applications **8**(4), 323–339 (2014). DOI 10.1007/s11761-013-0148-0
2. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE '06, pp. 201–210. ACM, New York, NY, USA (2006). DOI 10.1145/1173706.1173737
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013). DOI 10.1007/978-3-642-37521-7
4. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of International Software Product Line Conference (SPLC), *LNCS*, vol. 3714, pp. 7–20. Springer (2005). DOI 10.1007/11554844_3
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling stepwise refinement. IEEE Transactions on Software Engineering **30**, 355–371 (2004). DOI 10.1109/TSE.2004.23
6. ter Beek, M., Legay, A., Lluch Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Transactions on Software Engineering (2018). DOI: 10.1109/TSE.2018.2853726
7. Bettini, L., Damiani, F., Schaefer, I.: Compositional type checking of delta-oriented software product lines. Acta Informatica **50**(2), 77–122 (2013). DOI 10.1007/s00236-012-0173-z
8. Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. Theoretical Computer Science **455**, 2 – 30 (2012). DOI 10.1016/j.tcs.2012.01.031. International Colloquium on Theoretical Aspects of Computing 2010
9. Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., Hinchey, M.: An overview of dynamic software product line architectures and techniques: Observations from research and industry. Journal of Systems and Software **91**(0), 3 – 23 (2014). DOI 10.1016/j.jss.2013.12.038
10. Chesta, C., Damiani, F., Dobriakova, L., Guernieri, M., Martini, S., Nieke, M., Rodrigues, V., Schuster, S.: A toolchain for delta-oriented modeling of software product lines. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 9953, pp. 497–511.

Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-47169-3_40
11. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Formal Methods for Eternal Networked Software Systems, *Lecture Notes in Computer Science*, vol. 6659, pp. 417–457. Springer International Publishing (2011). DOI 10.1007/978-3-642-21455-4_13
12. Clements, P., Northrop, L.: Software Product Lines: Practices & Patterns. Addison Wesley Longman (2001)
13. Damiani, F., Faitelson, D., Gladisch, C., Tyszberowicz, S.: A novel model-based testing approach for software product lines. Software & Systems Modeling **16**(4), 1223–1251 (2017). DOI 10.1007/s10270-016-0516-2
14. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M.: A unified and formal programming model for deltas and traits. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10202, pp. 424–441. Springer (2017). DOI 10.1007/978-3-662-54494-5_25
15. Damiani, F., Lienhardt, M.: On type checking delta-oriented product lines. In: Integrated Formal Methods: 12th International Conference, iFM 2016, *LNCS*, vol. 9681, pp. 47–62. Springer (2016). DOI 10.1007/978-3-319-33693-0_4
16. Damiani, F., Lienhardt, M.: Refactoring delta-oriented product lines to achieve monotonicity. In: Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FM-SPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016., *EPTCS*, vol. 206, pp. 2–16 (2016). DOI 10.4204/EPTCS.206.2
17. Damiani, F., Lienhardt, M.: Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 9953, pp. 579–596 (2016). DOI 10.1007/978-3-319-47169-3_45
18. Damiani, F., Padovani, L., Schaefer, I., Seidl, C.: A core calculus for dynamic delta-oriented programming. Acta Informatica **55**(4), 269–307 (2018). DOI 10.1007/s00236-017-0293-6
19. Damiani, F., Schaefer, I.: Dynamic delta-oriented programming. In: Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11, pp. 34:1–34:8. ACM, New York, NY, USA (2011). DOI 10.1145/2019136.2019175
20. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An exploratory study of cloning in industrial software product lines. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, CSMR '13, pp. 25–34. IEEE Computer Society, Washington, DC, USA (2013). DOI 10.1109/CSMR.2013.13
21. Faitelson, D., Tyszberowicz, S.S.: Data refinement based testing. Int. J. Systems Assurance Engineering and Management **2**(2), 144–154 (2011). DOI 10.1007/s13198-011-0060-y
22. Fowler, M.: Refactoring: Improving the design of existing code. In: Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings, p. 256 (2002). DOI 10.1007/3-540-45672-4_31
23. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Evolving delta-oriented software product line architectures. In: R. Calinescu, D. Garlan (eds.) Large-Scale Complex IT Systems. Development, Operation and Management, pp. 183–208. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-34059-8_10
24. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer **41**(4), 93–95 (2008). DOI 10.1109/MC.2008.123
25. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231 – 274 (1987). DOI https://doi.org/10.1016/0167-6423(87)90035-9
26. Helvensteijn, M., Muschevici, R., Wong, P.Y.H.: Delta modeling in practice: a Fredhopper case study. In: Proc. of VAMOS'12, pp. 139–148. ACM (2012). DOI 10.1145/2110147.2110163
27. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS **23**(3), 396–450 (2001). DOI 10.1145/503502.503505
28. Itemis: Yakindu statechart tools. `http://www.itemis.com/en/yakindu/state-machine/`
29. Johansen, M.F., Haugen, O., Fleurey, F.: Properties of realistic feature models make combinatorial testing of product lines feasible. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 638–652. Springer-Verlag, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-24485-8_47
30. Johansen, M.F., Haugen, O., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. In: Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, pp. 46–55. ACM, New York, NY, USA (2012). DOI 10.1145/2362536.2362547
31. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Proc. of FTSCS 2016, *CCIS*, vol. 694, pp. 55–71. Springer (2017). DOI: 10.1007/978-3-319-53946-1_4
32. Kästner, C., Kuhlemann, M.: Automating feature-oriented refactoring of legacy applications. In: In ECOOP Workshop on Refactoring Tools (2007)
33. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP 2001— Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 2072, pp. 327–354. Springer (2001). DOI 10.1007/3-540-45337-7_18
34. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: DeltaJ 1.5: delta-oriented programming for Java. In: International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, pp. 63–74 (2014). DOI 10.1145/2647508.2647512
35. Kowal, M., Schulze, S., Schaefer, I.: Towards efficient spl testing by variant reduction. In: Proceedings of the 4th International Workshop on Variability & Composition, VariComp '13, pp. 1–6. ACM, New York, NY, USA (2013). DOI 10.1145/2451617.2451619
36. Krueger, C.: Eliminating the Adoption Barrier. IEEE Software **19**(4), 29–31 (2002). DOI 10.1109/MS.2002.1020284
37. Lienhardt, M., Clarke, D.: Conflict detection in delta-oriented programming. In: Leveraging Applications of

Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I, pp. 178–192 (2012). DOI 10.1007/978-3-642-34026-0_14
38. Lienhardt, M., Damiani, F., Testa, L., Turin, G.: On checking delta-oriented product lines of statecharts. Science of Computer Programming **166**, 3 – 34 (2018). DOI 10.1016/j.scico.2018.05.007
39. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: ICSE, pp. 112–121. ACM (2006). DOI 10.1145/1134285.1134303
40. Lochau, M., Goltz, U.: Feature interaction aware test case generation for embedded control systems. Electronic Notes in Theoretical Computer Science **264**(3), 37–52 (2010). DOI 10.1016/j.entcs.2010.12.013
41. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In: A.P. Black (ed.) ECOOP 2005 - Object-Oriented Programming, *LNCS*, vol. 3586, pp. 169–194. Springer (2005). DOI 10.1007/11531142_8
42. Monteiro, M.P., Fernandes, J.M.: Towards a catalogue of refactorings and code smells for aspectj. In: A. Rashid, M. Aksit (eds.) Transactions on Aspect-Oriented Software Development I, pp. 214–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). DOI 10.1007/11687061_7
43. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented Proof Theories and their Comparison, *Cambridge Tracts in Theoretical Computer Science*, vol. 46. Cambridge University Press (1998). DOI 10.1017/CBO9780511663079
44. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: J. Bosch, J. Lee (eds.) Software Product Lines: Going Beyond (SPLC 2010), *Lecture Notes in Computer Science*, vol. 6287, pp. 77–91. Springer Berlin Heidelberg (2010). DOI 10.1007/978-3-642-15579-6_6
45. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD'10, pp. 49–56. ACM, New York, NY, USA (2010). DOI 10.1145/1868688.1868696
46. Schulze, S., Richers, O., Schaefer, I.: Refactoring delta-oriented software product lines. In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development, AOSD '13, pp. 73–84. ACM, New York, NY, USA (2013). DOI 10.1145/2451436.2451446
47. Schulze, S., Thüm, T., Kuhlemann, M., Saake, G.: Variant-preserving refactoring in feature-oriented software product lines. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, pp. 73–81. ACM, New York, NY, USA (2012). DOI 10.1145/2110147.2110156
48. Wille, D., Runge, T., Seidl, C., Schulze, S.: Extractive software product line engineering using model-based delta module generation. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS'17, pp. 36–43. ACM, New York, NY, USA (2017). DOI 10.1145/3023956.3023957
49. Winkelmann, T., Koscielny, J., Seidl, C., Schuster, S., Damiani, F., Schaefer, I.: Parametric deltaj 1.5: Propagating feature attributes into implementation artifacts. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016., *CEUR Workshop Proceedings*, vol. 1559, pp. 40–54. CEUR-WS.org (2016)