Refactoring Delta-Oriented Product Lines to Achieve Monotonicity*

Ferruccio Damiani
University of Turin (Italy)
`ferruccio.damiani@di.unito.it`

Michael Lienhardt
University of Turin (Italy)
`michael.lienhardt@di.unito.it`

Delta-oriented programming (DOP) is a flexible transformational approach to implement software product lines. In delta-oriented product lines, variants are generated by applying operations contained in delta modules to a (possibly empty) 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 algorithms for refactoring a delta-oriented product line into monotonic form, i.e., either to contain add and modify operations only (monotonic increasing) or to contain remove and modify operations only (monotonic decreasing). Because of their simpler structure, monotonic delta-oriented product lines are easier to analyze. The algorithms are formalized by means of a core calculus for DOP of product lines of Java programs and their correctness and complexity are given.

1 Introduction

A *Software Product Line* (SPL) is a set of similar programs, called *variants*, that have a well documented variability and are generated from a common code base [4]. *Delta-Oriented Programming* (DOP) [14, 3] is a flexible and modular transformational approach to implement SPLs. 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 from which variants are obtained by applying program transformations, described by *delta modules*, that can add, remove or modify code. 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 module an *activation condition* over the features and specifies an *application ordering* between delta modules. DOP supports automated product derivation, i.e., once the features of a product are selected, the corresponding variant is generated by applying the activated delta modules to the base program according to the application ordering.

Delta modules are constructed from *delta operations* that can *add*, *modify* and *remove* content to and from the base program (e.g., for Java programs, a delta module can add, remove or modify classes interfaces, fields and methods). As pointed out in [15], such flexibility allows DOP to support *proactive* (i.e., planning all products in advance), *reactive* (i.e., developing an initial SPL comprising a limited set of products and evolving it as soon as new products are needed or new requirements arise), and *extractive* (i.e., gradually tranforming a set of existing programs into an SPL) SPL development [10]. DOP allows

---

*The authors of this paper are listed in alphabetical order. This work has been partially supported by project HyVar (`www.hyvar-project.eu`), which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644298; by ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`); and by Ateneo/CSP D16D15000360005 project RunVar.

Julia Rubin and Thomas Thüm (Eds.):
7th International Workshop on Formal Methods and Analysis
in Software Product Line Engineering (FMSPLE'16)
EPTCS 206, 2016, pp. 2–16, doi:10.4204/EPTCS.206.2

for quick SPL evolution and extension, as modifying or adding products/variants can straightforwardly be achieved by adding to the SPL new delta modules that modify, remove and add code on top of the original implementation of the SPL. However, a number of such SPL evolution and extension phases lead, almost ineluctably, to a multiplication of opposite add and remove operations, making the resulting SPL complex, difficult to understand and to analyze [16].

Refactoring [6] is an established technique to reduce complexity and to prevent the process of software aging, and consists of program transformations that change the internal structure of a program without altering its external (visible) behavior. Refactoring for DOP product lines, i.e. changing the internal structure of an SPL without changing its products/variants, has been investigated in [16]. There, a catalogue of refactoring algorithms and code smells is presented. Most of these refactorings are based on object-oriented refactorings [6]. In particular, the refactorings that refer to delta modules focus on a single delta module or a pair of delta modules.

In this paper, we propose two new refactoring algorithms to automatically eliminate opposite add and remove operations across the whole SPL, consequently reducing the overall complexity of the refactored SPL and making it easier to analyze. These algorithms are constructed around the notion of *monotonicity* where *increasing monotonic* SPL corresponds to only adding new content to the base program, while *decreasing monotonic* SPL corresponds to only removing content from the base program. These two notions of monotonicity are discussed in Section 5, where we propose several definitions with different versions of these concepts. The refactoring algorithms do not introduce code duplication in the AB of the refactored SPL and have at most quadratic complexity in space and time. We formalize the notions of monotonicity and the refactoring algorithms by means of IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ) [3], a core calculus for DOP product lines where variants are written in an imperative version of FEATHERWEIGHT JAVA (FJ) [8]. A prototypical implementation of the refactoring algorithms is available at [11].

Section 2 introduces our running example. Section 3 recalls IFΔJ. Section 4 introduces some auxiliary notations. Section 5 illustrates the notions of monotonicity, the refactoring algorithms, and their properties. Section 6 briefly discusses the related work and Section 7 concludes the paper.

2 Example

In order to illustrate the monotonicity concept and our refactoring algorithms, we use a variant of the *expression product line* (EPL) benchmark (see, e.g., [13, 3]). We consider the following grammar:

$$\mathsf{Exp} ::= \mathsf{Lit} \mid \mathsf{Add} \mid \mathsf{Neg} \qquad \mathsf{Lit} ::= \langle\mathsf{integers}\rangle \qquad \mathsf{Add} ::= \mathsf{Exp}\ "+"\ \mathsf{Exp} \qquad \mathsf{Neg} ::= "-"\ \mathsf{Exp}$$

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 int or as a literal expression.

2.1 The Feature Model

The functionalities 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 and Print are mandatory, while Add, Neg, Eval1 and Eval2 are optional. Moreover, as Eval1 and Eval2 define the same method, they are mutually exclusive. Figure 1 shows the feature model of the EPL represented as a feature diagram.

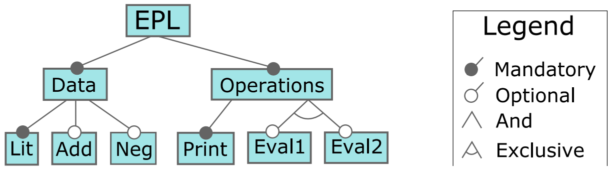

Figure 1: Expression Product Line: Feature Model

```
class Exp extends Object { // only used as a type
  String toString() { return null; }
}
class Lit extends Exp {
  int value;
  Lit setLit(int n) { value = n; return this; }
  String toString() { return value + ""; }
}
```

```
class Add extends Exp {
  Exp expr1;
  Exp expr2;
  Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
  String toString() { return expr1.toString() +
                              "␣+␣" + expr2.toString(); }
}
```

Figure 2: Base Program

2.2 The Artifact Base

Base Program. In our example, the EPL is constructed from the base program shown in Figure 2, which is the variant implementing features Lit, Add and Print. This program comprises the class `Exp`, the class `Lit` for literal expressions and the class `Add` for addition expressions. All these classes implement the `toString` method. Moreover, `Lit` and `Add` also have a setter method.

Implementing Feature Neg. Figure 3 presents the three delta modules (introduced by the keyword **delta**) that add the feature Neg to the base program. Namely: `DNeg` adds the class `Neg` with a simple setter; `DNegPrint` adds to class `Neg` the `toString` method (relevant for the Print feature); and `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. E.g., without applying `DOptionalPrint` both the following expressions

```
(new Add()).setAdd( new (Neg()).setNeg((new Lit()).setLit(3)), new (Lit()).setLit(5) ) // (−3) + 5
(new Neg()).setNeg( new (Add()).setAdd((new Lit()).setLit(3), new (Lit()).setLit(5)) ) // −(3+5)
```

would be printed as “-3+5”; while after applying `DOptionalPrint` the former is printed as “(-3+5)” and the latter is printed as “-(3+5)”. Delta module `DOptionalPrint` illustrates the usage of the special

```
delta DNeg {
  adds class Neg extends Exp {
    Exp expr;
    Neg setNeg(Exp a) { expr = a; return this; }
  }
}
delta DNegPrint {
  modifies Neg {
    adds String toString() {
     return "−" + expr.toString(); }
  }
}
```

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

Figure 3: Delta Modules for the Neg Feature

```
delta DLitEval1 {
  modifies Exp {
    adds int eval() { return 0; }
  }
  modifies Lit {
    adds int eval() { return value; }
  }
}
delta DAddEval1 {
  modifies Add {
    adds int eval() {
      return expr1.eval() + expr2.eval();
    }
  }
}
delta DNegEval1{
  modifies Neg {
    adds int eval() { return (−1) ∗ expr.eval(); }
  }
}
```

```
delta DLitEval2 {
  modifies Exp {
    adds Lit eval() { return null; }
  }
  modifies Lit {
    adds Lit eval() { return this; }
  }
}
delta DAddEval2 {
  modifies Add {
    adds Lit eval() {
      Lit res = exp1.eval();
      return res.setLit(res.value + exp2.eval()); }
  }
}
delta DNegEval2{
  modifies Neg {
    adds Lit eval() { Lit res = exp.eval();
      return res.setLit((−1) ∗ res.value); }
  }
}
```

Figure 4: Delta Modules for Features Eval1 (left) and Eval2 (right)

```
delta DremAdd { removes Add }
```

Figure 5: Delta Module for Removing the Add Feature

method `original` which allows here to call the original implementation of the method `toString`, and surround the resulting string with parenthesis.

Implementing Features Eval1 and Eval2. Figure 4 presents the delta modules that add the features Eval1 and Eval2 (on the left and on the right, respectively). The delta module `DLitEval1` (resp. `DLitEval2`) modifies the classes `Exp` and `Lit` 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 module `DAddEval1` (resp. `DAddEval2`) does the same operation on the `Add` class; and the delta module `DNegval1` (resp. `DANegEval2`) does the same operation on the `Neg` class.

Removing the Add Feature. If the feature Add is not selected, the generated variant must not contain the class `Add`. This is ensured by the delta module `DremAdd` in Figure 5 which removes the class `Add` from the program.

2.3 The Configuration Knowledge

The configuration knowledge specifies how variants are generated by i) specifying for which product (i.e., set of selected features) each delta module is activated, and ii) specifying a partial application order on the delta modules. Figure 6 presents the activation conditions and the partial order of the delta modules. The activation conditions and the partial order reflect the explanations about the delta modules of the EPL given in Section 2.2. For instance, the delta module `DNeg` is activated whenever the feature Neg is activated, the delta module `DremAdd` is activated whenever the feature Add is not selected, and the delta module `DOptionalPrint` is activated whenever both features Add and Neg are activated (recall that feature Print is mandatory).

Activations:

Delta Module	Activation
`DNeg`	Neg
`DNegPrint`	Neg ∧ Print
`DOptionalPrint`	Neg ∧ Add

Delta Module	Activation
`DLitEval1`	Eval1
`DAddEval1`	Eval1 ∧ Add
`DNegEval1`	Neg ∧ Eval1

Delta Module	Activation
`DLitEval2`	Eval2
`DAddEval2`	Eval2 ∧ Add
`DNegEval2`	Neg ∧ Eval2
`DremAdd`	¬Add

Order: $\mathtt{DNeg} <_L \{\, \mathtt{DNegPrint}, \mathtt{DOptionalPrint} \,\}$
$<_L \{\, \mathtt{DLitEval1}, \mathtt{DAddEval1}, \mathtt{DNegEval1} \,\} <_L \{\, \mathtt{DLitEval2}, \mathtt{DAddEval2}, \mathtt{DNegEval2} \,\} <_L \mathtt{DremAdd}$

Figure 6: Expression Product Line: Configuration Knowledge

$$
\begin{array}{lll}
P & ::= \overline{CD} & \text{Program}\\
CD & ::= \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{C}\ \{\ \overline{AD}\ \} & \text{Class}\\
AD & ::= FD \ \mid\ MD & \text{Attribute (Field or Method)}\\
FD & ::= \mathtt{C}\ \mathtt{f} & \text{Field}\\
MD & ::= \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\ \{\textbf{return}\ e;\} & \text{Method}\\
e & ::= \mathtt{x} \ \mid\ e.\mathtt{f} \ \mid\ e.\mathtt{m}(\overline{e}) \ \mid\ \textbf{new}\ \mathtt{C}() \ \mid\ (\mathtt{C})e \ \mid\ e.\mathtt{f} = e \ \mid\ \textbf{null} & \text{Expression}
\end{array}
$$

$$
\begin{array}{lll}
L & ::= P\ \ \overline{\Delta}\ \ FM\ \ CK & \text{Product Line}\\
\Delta & ::= \textbf{delta}\ \mathtt{d}\ \{\ \overline{CO}\ \} & \text{Delta Module}\\
CO & ::= \textbf{adds}\ CD \ \mid\ \textbf{removes}\ \mathtt{C} \ \mid\ \textbf{modifies}\ \mathtt{C}\ [\textbf{extends}\ \mathtt{C}']\ \{\ \overline{AO}\ \} & \text{Class Operation}\\
AO & ::= \textbf{adds}\ AD \ \mid\ \textbf{modifies}\ MD \ \mid\ \textbf{removes}\ \mathtt{a} & \text{Attribute Operation}
\end{array}
$$

Figure 7: Synax of IFJ (top) and IFΔJ (bottom)

Following [3], the partial order is specified as a total order on a partition of the set of delta modules. The partial order must ensure that the variants of the EPL can be generated. Therefore, it states that the delta modules `DNeg` (that adds the class `Neg`) must be applied before `DNegPrint`, `DNegEval1` and `DNegEval2` (that modify class `Neg`). The partial order also ensures that, independently from the activation conditions, the delta modules occurring in the same partition perform disjoint delta operations (thus guranteeing that applying any subset of them in any possible order always produces the same transformation)—this guarantees that the product line is unambiguous (i.e., applying the activated delta modules in any possible total order that respects the application order produces the same variant). Therefore, the delta modules for feature Eval1 and the delta modules for feature Eval2 are put in two different parts; and the delta module `DremAdd` (that removes the class `Add`) is applied after `DAddEval1`, `DAddEval2` and `DOptionalPrint` (that modify class `Add`).

3 The IFΔJ Calculus

In this section we briefly recall the IFΔJ [3] core calculus for DOP. We present the calculus in two steps: (i) we introduce the IFJ calculus, which is an imperative version of FJ [8]; and (ii) we introduce the constructs for variability on top of it. The full descriprion of IFΔJ is given in [3], where a type-cheching technique for ensuring type soundness of all variants is presented. The version of IFΔJ presented in this paper is indeed a slight extension of the one presented in [3]: the AB contains also an IFJ program outside of any delta module. This makes the IFΔJ syntax a direct extension of the IFJ syntax.

The abstract syntax of IFJ is presented in Figure 7 (top). Following [8], we use the overline notation for (possibly empty) sequences of elements: for instance $\overline{e}$ stands for a sequence of expressions. Variables `x` include the special variable `this` (implicitly bound in any method declaration *MD*), which may not be used as the name of a formal parameter of a method. A program *P* is a sequence of class declarations $\overline{CD}$. A class declaration **class** `C` **extends** $\mathtt{C}'$ $\{\ \overline{AD}\ \}$ comprises the name `C` of the class, the name $\mathtt{C}'$ of the superclass (which must always be specified, even if it is the built-in class `Object`), and a list of field and method declarations $\overline{AD}$. 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**. The subtyping relation $<:$ on classes, which is the reflexive and transitive closure of the immediate subclass relation (given by the **extends** clauses in class declarations), is supposed to be acyclic.

The abstract syntax of the language IFΔJ is given in Figure 7 (bottom). An IFΔJ SPL L comprises: a possibly empty or incomplete IFJ program P; a set of delta modules $\overline{\Delta}$ that, together with the base program P, represents the artifact base; a feature model FM specifying the features and the products of the SPL; and a configuration knowledge CK (i.e., the ordering between delta modules and their activation conditions).

To simplify the presentation, we do not give a syntactic description of FM nor of CK and we rely on getter functions as follows: L.`features` is the set of features; L.`products` specifies the products (i.e., a subset of the power set $2^{L.\texttt{features}}$); L.`activation` maps each delta module name `d` to its activation condition; and L.`order` (or $<_L$, for short) is the application ordering between the delta modules.

A delta module declaration Δ comprises the name `d` of the delta module and class operations $\overline{CO}$ representing the transformations performed when the delta module 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 name* `a` is either a field name `f` or a method name `m`. 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 `original`, which is implicitly bound to the previous implementation of the method and may not be used as the name of a method.

The *projection* of a product line on a subset of its products is obtained by restricting L.`products` to describe only the products in the subset and by dropping delta modules that are never activated.

Example 1. *For instance, the AB of the projection of the EPL on the products without feature Neg is obtained by dropping the delta modules* `DNeg`, `DNegPrint` *and* `DOptionalPrint`*; and the AB of the projection of the EPL on the products without feature Eval2 is obtained by dropping the delta modules* `DLitEval2`, `DAddEval2` *and* `DNegEval2`.

4 Auxiliary Notations

In this section we introduce some auxiliary notations that will be used in Section 5. Our first notation relates the **modifies** operators on methods to the concept of monotonicity. Indeed, in general **modifies** on methods is not monotonic: the body of the method is replaced by some code that can be entirely different. However, we can distinguish two cases in which **modifies** can be considered monotonic: when it calls `original`, the generated variant contains the original body of the method, and so **modifies** can be considered *increasing monotonic*; when the body of the method is *voided* (i.e., it is replaced by **return null**) **modifies** can be considered *decreasing monotonic*.

Notation 1 (**wraps** and **voids**). *Let* **wraps** *denote a* **modifies** *operation on method that calls* `original`, *and* **voids** *denote a* **modifies** *operation that removes the content of a method:* **voids** `m` *corresponds to* **modifies** `C m(···)` {**return null**}.

The goal of the two following notations is to unify delta operations on classes and on attributes in a single model, in order to manage uniformly these two kind of operations in our refactoring algorithms. Using these notations simplifies the description of our refactoring algorithms.

Notation 2. *A* reference, *written ρ, is either a class name* `C` *or a qualified attribute name* `C.a` *and we write $\rho \le \rho'$ if $\rho = \rho'$ or if ρ is a prefix of ρ'. By abuse of notation, we also consider the* **extends** *clause as an attribute of its class, and consider* `C.`**extends** *as a valid reference.*

Notation 3. *We abstract a delta module by a set of* Abstract Delta Operations *(ADO) which are triplets* $(\textbf{dok}, \rho, \mathrm{D})$ *where: i)* **dok** *is a delta operation keyword (***adds***,* **removes** *or* **modifies***), ii) ρ is the reference on which* **dok** *is applied, iii)* D *is the data associated with this operations, and iv) if* $\textbf{dok} = \textbf{modifies}$ *then ρ is not a class name. Given an ADO* `o`*, we denote its operator as* `o.`**dok***, its reference as* $\mathtt{o}.\rho$ *and its data as* $\mathtt{o}.\mathrm{D}$.

These two notations are ilustrated by the following examples. In particular, the first example shows that a **modifies** operation on a class `C` that contains only **adds** operations on attributes is represented by the set of ADOs containing only the **adds** operations: the **modifies** `C` operation is only a syntactic construction to introduce these **adds** operations and is not included in our representation.

Example 2. *The delta module* `DLitEval2` *in Figure 4 that modifies classes* `Exp` *and* `Lit` *by adding a method* `eval` *to each of them, is modeled with only two ADOs:*

$$(\textbf{adds},\quad \mathsf{Exp.eval},\quad \mathsf{Lit\ eval()}\ \{\ \textbf{return null};\ \})\qquad \textit{and}\qquad (\textbf{adds},\quad \mathsf{Lit.eval},\quad \mathsf{Lit\ eval()}\ \{\ \textbf{return}\ \mathsf{this};\ \})$$

These ADOs model the addition of the `eval` *methods, the modification of classes* `Exp` *and* `Lit` *being implicit as* `Exp` *(resp.* `Lit`*) is a prefix of* `Exp.eval` *(resp.* `Lit.eval`*).*

Example 3. *The delta module* `DOptionalPrint` *in Figure 3 that modifies the class* `Add` *by modifying the method* `toString`*, is modeled with only one ADO:*

$$(\textbf{modifies},\quad \mathsf{Add.toString},\quad \mathsf{String\ toString()}\ \{\ \textbf{return}\ \text{``(''} + \mathsf{original()} + \text{``)''};\ \})$$

Example 4. *Note that, according to Definition 3, the projection of the EPL on the products without feature Neg does not contain* **modifies** *operations.*

Our last notations are used to iterate over delta modules: first, we present the notations to get a set of delta module names, then we present the notations to order such a set so to iterate over it in a **for** loop.

Notation 4. *The set of delta module names declared in L is denoted as* $\mathtt{dm}(L)$*. When L is clear from the context, we write* $\mathtt{before}(\mathtt{d})$ *the set of delta module names that are before* `d` *for* L`.order`.

Notation 5. *Given a set of delta names $S = \{\mathtt{d}_i \mid i \in I\}$, we denote $\uparrow\!S$ (resp. $\downarrow\!S$) a sequence $(\mathtt{d}_{i_1}, \ldots, \mathtt{d}_{i_n})$ of all the names in S that respects the partial order (resp. the partial order opposite from the one) specified by* L`.order`.

5 Monotonicity and Refactoring Algorithms

In the introduction, we pointed out that the flexibility provided by delta operations, being very useful for easily constructing SPLs, can lead to unnecessary complexity with many adding and removing operations cancelling each other. Monotonicity is a natural approach to lower such complexity as it forbids opposite adding and removing operations: informally, *increasing monotonicity* is constructing a variant only by adding new content to the base program and is in principle similar to *Feature-Oriented Programming* (FOP) [2];[1] on the other hand, *decreasing monotonicity* is constructing it only by removing content from the base program and share similarities with annotative approaches (see, e.g., [5, 9]).

---

[1] As pointed out in [15], DOP is a generalization of FOP: the AB of a FOP product line consists of a set of *feature modules* which are delta modules that correspond one-to-one to features and do not contain remove operations.

Section 5.1 focuses on increasing monotonicity: it formalizes and motivates different levels of purity for it, then presents a refactoring algorithm transforming an SPL into an increasing monotonic equivalent and illustrates it on the EPL example. Section 5.2 formalizes decreasing monotonicity, presents a refactoring algorithm and its application to the EPL. Section 5.3 gives correctness and complexity of the refactoring algorithms.

5.1 Increasing Monotonicity

Before presenting the first refactoring algorithm, we gradually introduce three notions of increasing monotonicity, from the most intuitive one, called *strictly-increasing*, to the most flexible one, called *pseudo-increasing*. Depending on the properties of the input SPL, the algorithm can produce SPLs corresponding to any of the three notions. A first intuitive notion of increasing monotonicity is only to allow **adds** operations:

Definition 1 (Strictly-increasing monotonic)**.** *An SPL is* strictly-increasing monotonic *iff it only contains* **adds** *operations.*

Note that this notion is quite restrictive, as it does not allow the extension of method implementation, or the modification of the **extends** clause of a class, two operations possible in FOP. The following more liberal notion allows to increase the body of existing methods by using the **modifies** operator by always calling `original`. Still, it does not include the modification of the **extends** clause of a class present in FOP.

Definition 2 (Increasing Monotonic)**.** *An SPL is* increasing monotonic *iff it only contains* **adds** *and* **wraps** *operations.*

The last notion, which is a generalization of FOP, is to allow **modifies** also to modify the **extends** clause of a class and to replace the implementation of a method, leaving only **removes** as a forbidden operation:

Definition 3 (Pseudo-increasing monotonic)**.** *An SPL is* pseudo-increasing monotonic *iff it does not contain* **removes** *operations.*

We have qualified the above notion as *pseudo-*, since it allows delta modules to replace the **extends** clause of a class and to remove or entirely replace content from the body of method definitions. Thus, it does not reflect the informal definition of increasing monotonicity given at the beginning of Section 5.

5.1.1 Increasing Monotonicity Refactoring Algorithm

The refactoring algorithm, presented in Figure 8, transforms its input DOP product line L by eliminating all **removes** operations and without eliminating or introducing new **modifies** operations. Therefore, the refactored SPL is

- strictly-increasing, if L does not contain **modifies** operations;
- increasing, if all the **modifies** operations in L are **wraps** operations; and
- pseudo-increasing, otherwise.

Note that the algorithm may turn an existing delta module into an empty delta module which can then can be removed by a straightforward algorithm (see [16]).

To illustrate how the refactoring algorithm works, consider a delta module `d` containing a removal operation on an element ρ (either a class or an attribute). This operation would be applied only when `d` is activated, and would remove all declarations (and modification) of ρ that are done *before* the application

```
1   Delta Module Name: $\mathtt{d}_1$, $\mathtt{d}_2$;
2   Operation: $\mathtt{o}_1$, $\mathtt{o}_2$;
3   Set of Delta Module Name: $S$;
4
5   refactor($L$) =
6     for $\mathtt{d}_1 \in \uparrow\mathtt{dm}(L)$ do
7       for $\mathtt{o}_1 \in L(\mathtt{d}_1)$ do
8         if($\mathtt{o}_1.\mathbf{dok} = \mathbf{removes}$)
9           $L(\mathtt{d}_1) \leftarrow L(\mathtt{d}_1) \setminus \mathtt{o}_1$
10          manageOperation()
11        fi
12      done
13    done;
14
15  manageOperation() =
16    $S \leftarrow \emptyset$
17    for $\mathtt{d}_2 \in \downarrow\mathtt{before}(\mathtt{d}_1)$ do
18      for $\mathtt{o}_2 \in L(\mathtt{d}_2)$ do
19        if($\mathtt{o}_1.\rho \leq \mathtt{o}_2.\rho$) mergeOperations() fi
20      done
21    done
22    mergeToBase();
23  mergeOperations() =
24    $S \leftarrow S \cup \{\mathtt{d}_2\}$
25    $L(\mathtt{d}_2) \leftarrow L(\mathtt{d}_2) \setminus \mathtt{o}_2$
26    if($L(\mathtt{d}_2) = \{\ \}$) $L \leftarrow L \setminus \mathtt{d}_2$ fi
27    $L \leftarrow L + \mathtt{d}$ fresh with {
28      $L(\mathtt{d}) \leftarrow \{\ \mathtt{o}_2\ \}$
29      $L.\mathtt{activation}(\mathtt{d}) \leftarrow \mathtt{d}_2 \wedge \neg\mathtt{d}_1$
30      $L.\mathtt{order}(\mathtt{d}) \leftarrow L\mathtt{order}(\mathtt{d}_2)$
31    }
32
33  mergeToBase() =
34    $\mathrm{D} \leftarrow L.P(\mathtt{o}_1.\rho)$
35    if($\mathrm{D} \neq \perp$)
36      $L.P \leftarrow \mathtt{apply}(\mathtt{o}_1, L.P)$
37      $L \leftarrow L + \mathtt{d}$ fresh with {
38        $L(\mathtt{d}) \leftarrow \{\ (\ \mathbf{adds},\ \mathtt{o}_1.\rho,\ \mathrm{D}\ )\ \}$
39        $L.\mathtt{activation}(\mathtt{d}) \leftarrow \neg\mathtt{d}_1$
40        $L.\mathtt{order}(\mathtt{d}) \leftarrow$ before($S$)
41      }
42    fi;
```

Figure 8: Refactoring Algorithm for Increasing Monotonic SPL

of `d`. Hence, to cancel this removal operation, we can simply transform the SPL so that ρ is never declared before `d` and when it is activated.

The algorithm is structured in four functions with four global variables. The main function of our algorithm is `refactor` which takes the SPL to refactor as parameter. This function looks in order at all the delta modules and when finding a **removes** operation $\mathtt{o}_1$ inside a delta module $\mathtt{d}_1$, it cancels it from $\mathtt{d}_1$ and calls the `manageOperation` function. The goal of the `manageOperation` function is to transform the SPL for the $\mathtt{o}_1$ operation as described before. It is structured in two parts. First, it looks in order at all the delta operations applied before $\mathtt{d}_1$, and upon finding an operation $\mathtt{o}_2$ in a delta module $\mathtt{d}_2$ that manipulates $\mathtt{o}_1.\rho$, it calls `mergeOperation` which extracts that operation from $\mathtt{d}_2$ and changes the application condition of $\mathtt{o}_2$ (using a freshly created delta module `d`) so it is executed only when $\mathtt{o}_1$ would not be executed. Second, it calls `mergeToBase` which looks if the element removed by $\mathtt{o}_1$ is declared in the base program, and if so, extracts it from the base program into a fresh opposite delta module `d` that is activated only when $\mathtt{o}_1$ would not be executed. The addition of this new delta module is done in lines 37–41 where we state that L is changed by adding a fresh delta module `d` with the following characteristics: its set of ADO $L(\mathtt{d})$ is the singleton $(\mathbf{adds}, \mathtt{o}_1.\rho, \mathrm{D})$ that adds $\mathtt{o}_1.\rho$ again to the base program; its activation condition $L.\mathtt{activation}(\mathtt{d})$ is the opposite of $\mathtt{d}_1$; and its ordering $L.\mathtt{order}(\mathtt{d})$ states that it must be applied before all the delta modules in S.

There are three subtleties in this algorithm. First, to deal with the fact that removing a class also removes all its attributes, the condition in line 19 is "$\mathtt{o}_1.\rho \leq \mathtt{o}_2.\rho$" meaning that: if $\mathtt{o}_1$ removes a class `C`, then previous additions and modifications of `C` and its attributes will be changed with `mergeOperation`. Second, in line 26, empty delta modules are eliminated to avoid creating too much of them. Third, we compute in S the set of all delta modules manipulating $\mathtt{o}_1.\rho$ before $\mathtt{d}_1$ to set the order relation of the delta module created in the `mergeToBase` function.

```
delta DNotDremAdd {
  adds class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add setAdd(Exp a, Exp b) {
      expr1 = a; expr2 = b; return this; }
    String toString() { return expr1.toString()
      + "␣+␣" + expr2.toString(); }
} }
delta DOptionalPrint_DremAdd {
   modifies Add {
      modifies String toString() {
       return "(" + original() + ")"; }
} }
```

```
delta DAddEval1_DremAdd {
   modifies Add {
      adds int eval() {
         return expr1.eval() + expr2.eval();
      }
} }
delta DAddEval2_DremAdd {
   modifies Add {
      adds Lit eval() {
         Lit res = exp1.eval();
         return res.setLit(res.value + exp2.eval()); }
} }
```

Figure 9: Delta Modules of the EPL Changed by the Increasing Refactoring Algorithm

5.1.2 Example: Refactoring the EPL into Increasing Monotonicity

We applied our implementation of this algorithm on the EPL given in Section 2. It contains only one **removes** operation, in the `DremAdd` delta module, removing the Add class. Thus, by construction of our algorithm, only the delta modules `DAddEval1`, `DAddEval2`, `DOptionalPrint` and the base program, that modify and declare the Add class (respectively), are changed by the refactoring process.

Let us illustrate the modification done on the delta modules by considering `DAddEval1`: the function `mergeOperations` extract the only operation inside this delta module (line 25), removes `DAddEval1` as it is now empty (line 26), and then basically recreates it (line 27), with the activation condition extended with $\neg$`DremAdd`, corresponding to `Add`. Hence, the delta modules are simply renamed by the algorithm. However, the base program is changed by the function `mergeToBase` which removes the class Add from it, and creates a new delta module reintroducing that class with the activation condition $\neg$`DremAdd` which corresponds to `Add`.

The modified delta modules are shown in Figure 9. The modified base program, which is not shown, is obtained from the original base program (see Figure 2) by dropping the declaration of class `Add`. Note that, since all the **modifies** operations of the original SPL were **wraps** operations, the refactored SPL is increasing monotonic. On the other hand, since the projection of the original EPL on the products without feature Neg does not contain **modifies** operations (see Example 4 in Section 4), its increasing monotonic refactoring would produce a strict-increasing product line.

5.2 Decreasing Monotonicity

Like for increasing monotonicity, we introduce several levels of purity for decreasing monotonicity before presenting the refactoring algorithm. Straightforward adaptations of Definition 1, 2 and 3 lead to the following definitions of strictly-decreasing, decreasing and pseudo-decreasing monotonicity.

Definition 4 (Strictly-decreasing monotonic)**.** *An SPL is* strictly-decreasing monotonic *iff it only contains* **removes** *operations.*

Definition 5 (Decreasing Monotonic)**.** *An SPL is* decreasing monotonic *iff it only contains* **removes** *operations and* **voids** *operations.*

Definition 6 (Pseudo-decreasing monotonic)**.** *An SPL is* pseudo-decreasing monotonic *iff it only contains* **removes** *and* **modifies** *operations.*

Unfortunately, the three above notions suffer of a major drawback: not all product lines can be expressed by following their prescriptions. For instance, in order to conform to any of Definition 4, 5 and 6, the base program of the EPL (cf. Section 2) must contain the class declaration

```
class Exp extends Object {
    String toString() { return null; }
    Lit eval() { return null; }
    int eval() { return 0; }
}
```

that contains two method declarations with same signature eval() and therefore is not valid in Java. In order to overcome this drawback, we introduce the following notation to express the notion of "readding" (i.e., to remove and to immediately add) an attribute.

Notation 6 (readds). *Let* $(\textsf{readds}, \rho, \mathrm{D})$ *denotes the sequence of removing the attribute* ρ*, and then performing* $(\textsf{adds}, \rho, \mathrm{D})$.

We can now give the definitions of read-strictly-decreasing, readd-decreasing and read-pseudo-decreasing monotonicity that does not suffer of the above drawback.

Definition 7 (Readd-strictly-decreasing monotonic)**.** *An SPL is* readd-strictly-decreasing monotonic *iff it only contains* **readds** *and* **removes** *operations.*

Definition 8 (Readd-decreasing monotonic)**.** *An SPL is* readd-decreasing monotonic *iff it only contains* **readds** *operations,* **removes** *operations and* **voids** *operations.*

Definition 9 (Readd-pseudo-decreasing monotonic)**.** *An SPL is* readd-pseudo-decreasing monotonic *iff it only contains* **readds**, **removes** *and* **modifies** *operations.*

5.2.1 Decreasing Monotonicity Refactoring Algorithm

Our algorithm, presented in Figure 10, refactors a DOP product line *L* by eliminating all **adds** operations and without eliminating or introducing new **modifies** operations. Therefore, the refactored SPL is

- readd-strictly-decreasing if *L* does not contain **modifies** operations;
- readd-decreasing if all the **modifies** operations in *L* are **voids** operations; and
- readd-pseudo-decreasing, otherwise.

The decreasing monotonic refactoring algorithm may introduce empty new delta modules. As pointed out in the discussion at the beginning of Section 5.1.1, empty delta modules can be removed from the refactored product line by a straightforward algorithm. Moreover, if each class/attribute is introduced (i.e., either declared in the base program or added by a delta module) only once, then decreasing monotonic refactoring does not introduce **readds** operations.

The structure of this refactoring algorithm is similar to the one to get increasing monotonicity: the main function `refactor` takes as parameter the SPL to refactor, and iterates over all the delta modules to find an **adds** operator to remove. Upon finding an operation $\mathtt{o}_1$ with an **adds** operator in a delta module $\mathtt{d}_1$, the function `manageOperation` is called. This function, like for the increasing refactoring algorithm, is structured in two parts. First, it looks in order at all the delta operations applied before $\mathtt{d}_1$, and upon finding an operation $\mathtt{o}_2$ in a delta module $\mathtt{d}_2$ that manipulates $\mathtt{o}_1.\rho$ with a **removes** operator, it calls `mergeOperation` which extracts that operation from $\mathtt{d}_2$ and update the application condition of $\mathtt{o}_2$ as done in the other algorithm. Second, it calls `mergeToBase` which integrates the operations $\mathtt{o}_1$ in the base program as follows: first, it completes the base program with all the declarations introduced in $\mathtt{o}_1$ that was missing from it; second, it creates a new delta module `d` that readds (see Definition 6) all the declarations originally done in the base program by the ones done in $\mathtt{o}_1$; finally, it creates a new delta module $\mathtt{d}'$ opposite to $\mathtt{o}_1$ that removes all the declarations done in $\mathtt{o}_1$ if these operations would not be executed. For the creation of these delta modules in lines 35–43, we use the following notations: $\mathrm{dom}(\mathtt{o})$ is the set of references that are declared in that operations, and $\mathtt{o}(\rho)$ is the data D associated to ρ in `o`. For instance, with `o` being the **adds** operation in the `DNeg` delta module, we have

```
1   Delta Module Name $\mathtt{d}_1, \mathtt{d}_2$;
2   Operation $\mathtt{o}_1, \mathtt{o}_2$;
3
4   refactor($L$) =
5     for module $\mathtt{d}_1 \in \uparrow\mathtt{dm}(L)$ do
6       for $\mathtt{o}_1 \in L(\mathtt{d}_1)$ do
7         if($\mathtt{o}_1.\mathbf{dok} = \mathbf{adds}$)
8           $L(\mathtt{d}_1) \leftarrow L(\mathtt{d}_1) \setminus \mathtt{o}_1$
9           manageOperation()
10        fi
11      done
12    done;
13
14  manageOperation() =
15    for module $\mathtt{d}_2 \in \downarrow\mathtt{before}(\mathtt{d}_1)$ do
16      for $\mathtt{o}_2 \in L(\mathtt{d}_2)$ do
17        if(($\mathtt{o}_2.\rho \in \mathrm{dom}(\mathtt{o}_1)$) & ($\mathtt{o}_2.\mathbf{dok} = \mathbf{removes}$))
18          mergeOperations()
19        fi
20      done
21    done
22    mergeToBase();
23  mergeOperations() =
24    $L(\mathtt{d}_2) \leftarrow L(\mathtt{d}_2) \setminus \mathtt{o}_2$
25    if($L(\mathtt{d}_2) = \emptyset$) $L \leftarrow L \setminus \mathtt{d}_2$ fi
26    $L \leftarrow L + \mathtt{d}$ fresh with {
27      $L(\mathtt{d}) \leftarrow \{\ \mathtt{o}_2\ \}$
28      $L.\mathtt{activation}(\mathtt{d}) \leftarrow \mathtt{d}_2 \wedge \neg \mathtt{d}_1$
29      $L.\mathtt{order}(\mathtt{d}) \leftarrow L.\mathtt{order}(\mathtt{d}_2)$
30    };
31
32  mergeToBase() =
33    Set of reference: $S \leftarrow \mathrm{dom}(L.P)$
34    $L.P \leftarrow L.P \cup \{\rho\ \mathrm{D} \mid (\mathbf{adds}, \rho, \mathrm{D}) \in \mathtt{o}_1 \wedge \rho \notin S\}$;
35    $L \leftarrow L + \mathtt{d}$ fresh with {
36      $L(\mathtt{d}) \leftarrow \{\ (\mathbf{readds}, \mathtt{C.a}, \mathtt{o}_1(\rho)) \mid \mathtt{C.a} \in \mathrm{dom}(\mathtt{o}_1) \cap S\}$
37      $L.\mathtt{activation}(\mathtt{d}) \leftarrow \mathtt{d}_1$
38      $L.\mathtt{order}(\mathtt{d}) \leftarrow L.\mathtt{order}(\mathtt{d}_1)$
39    } + $\mathtt{d}'$ fresh with {
40      $L(\mathtt{d}') \leftarrow \{\ (\mathbf{removes}, \rho, \emptyset) \mid \rho \in \mathrm{dom}(\mathtt{o}_1) \setminus S\}$
41      $L.\mathtt{activation}(\mathtt{d}') \leftarrow \neg \mathtt{d}_1$
42      $L.\mathtt{order}(\mathtt{d}') \leftarrow L.\mathtt{order}(\mathtt{d}_1)$
43    };
```

Figure 10: Refactoring Algorithm for Decreasing Monotonic SPL

$\mathrm{dom}(\mathtt{o}) = \{\mathtt{Neg}, \mathtt{Neg.expr}, \mathtt{Neg.setNeg}\}$ and, e.g., $\mathtt{o}(\mathtt{Neg.expr}) = (\mathtt{Exp\ expr})$

There are two subtleties in this algorithm. First, it can occur that before an **adds** operation adding a class `C`, removal operations can be applied on the *attributes* of `C`, and so, the condition in line 17 "$\mathtt{o}_2.\rho \in \mathrm{dom}(\mathtt{o}_1)$" captures all possible attributes of $\mathtt{o}_1.\rho$. Second, in line 36, we only readd attributes, not classes, to ensure that the base program contains every elements declared in the SPL. Note also that in this example, there is no need of a set S to define the order of the delta modules created in `mergeToBase`: the order simply is the one of the original $\mathtt{d}_1$ delta module.

5.2.2 Example: Refactoring the EPL into Decreasing Monotonicity

We applied this refactoring algorithm to the EPL example. All its delta modules but `DremAdd` and `DOptionalPrint` add new content to the base program, and all of them are modified by the refactoring as follows: they are emptied out by the `refactor` function which removes the **adds** operations, that are then reintroduced to the SPL by the `mergeToBase` in the base program with few new delta modules. The structure of the resulting SPL is presented in Figure 11—it contains 8 empty delta modules (lines 27, 29, 31, 33, 38, 41, 44 and 47), which can be straightforwardly removed. The left part of Figure 11 contains the new base program which now contains all the elements declared in the SPL: the class Neg as well as the attributes `toString` and `eval` are declared in the base program. Note that as the delta modules implementing the Eval1 feature are before the ones implementing the Eval2 feature, the new base program contains the Eval1 version of the `eval` methods. The right part of Figure 11 presents the newly added delta modules. The names of these delta modules are constructed in two parts: first the operation they perform, and then the delta module that created them. For instance, `DremNeg_DNeg` is the removing delta module created in the `mergeToBase` function from the `DNeg` delta module: it removes the Neg class when the feature Neg is not selected. The second delta module `DremNegToString_DNegPrint` is the delta module removing the method `Neg.toString` when neither Neg nor Print are selected. The second

```
 1  class Exp extends Object {
 2    String toString() { return ""; }
 3    int eval() { ... }
 4  }
 5  class Lit extends Exp {
 6    int value;
 7    Lit setLit(int n) { ... }
 8    String toString() { ... }
 9    int eval() { ... }
10  }
11  class Add extends Exp {
12    Exp expr1;
13    Exp expr2;
14    Add setAdd(Exp a, Exp b) { ... }
15    String toString() { ... }
16    int eval() { ... }
17  }
18  class Neg extends Exp {
19    Exp expr;
20    Neg setNeg(Exp a) { ... }
21    String toString() { ... }
22    int eval() { ... }
23  }
24  DremNeg_DNeg { removes Neg }
25  DremNegToString_DNegPrint { modifies class Neg { removes toString } }
26
27  DreaddNegEval_DNegEval1 { }
28  DremNegEval_DNegEval1 { modifies class Neg { removes eval } }
29  DreaddExpEval_DLitEval1 { }
30  DremExpEval_DLitEval1 { modifies class Exp { removes eval } }
31  DreaddLitEval_DLitEval1 { }
32  DremLitEval_DLitEval1 { modifies class Lit { removes eval } }
33  DreaddAddEval_DAddEval1 { }
34  DremAddEval_DAddEval1 { modifies class Add { removes eval } }
35
36  DreaddNegEval_DNegEval2 {
37    modifies class Neg { readds Lit eval() { ... } } }
38  DremNegEval_DNegEval2 { }
39  DreaddExpEval_DLitEval2 {
40    modifies class Exp { readds Lit eval() { ... } } }
41  DremExpEval_DLitEval2 { }
42  DreaddLitEval_DLitEval2 {
43    modifies class Lit { readds Lit eval() { ... } } }
44  DremLitEval_DLitEval2 { }
45  DreaddAddEval_DAddEval2 {
46    modifies class Add { readds Lit eval() { ... } } }
47  DremAddEval_DAddEval2 { }
```

Figure 11: EPL Modified by the Decreasing Refactor Algorithm

set of delta modules (from line 27 to 34) corresponds to the integrations of the Eval1 feature in the base program. For instance, `DreaddNegEval_DNegEval1` is the `d` delta module created by the `mergeToBase` function (line 35 in Figure 10), and does not contain any operations as the base program did not originally contain the `eval` method; `DremNegEval_DNegEval1` is the `d`′ delta module created by the `mergeToBase` function (line 39 in Figure 10), and removes the `Neg.eval` method when the feature Eval1 or Neg is not selected. The last set of delta modules (from line 36 to 47) corresponds to the integrations of the Eval2 feature in the base program. As when including this feature in the base program, the delta modules for Eval1 already have been integrated, the *readding* delta modules contains the implementation of the Eval2 version of the `eval` method; and on the opposite, the *removing* delta modules are empty.

Note that, since the original SPL contains method **modifies** operations that are not **voids**, the refactored SPL is readds-pseudo-decreasing monotonic.On the other hand, since in the projection of the original EPL on the products without feature Eval2 each class/attribute is added only once (see Example 1 in Section 3), its decreasing monotonic refactoring would produce a pseudo-decreasing product line.

5.3 Properties

We finally present the main properties of these two refactoring algorithms. As they both share the same characteristics, we state our theorems for both of them.

Theorem 1 (Correctness)**.** *Applying one of the* `refactor` *algorithms on one SPL L is a monotonic SPL that have the same products and variants as L.*

Proof (sketch). Let us consider the increasing version of the `refactor` algorithm (proving the result for the decreasing version is similar), and let us denote L' as $\texttt{refactor}(L)$. The fact that L' is monotonic is a direct consequence of the algorithm iterating over all delta operations and deleting all the **removes** operations. The fact that L' has the same products as L is a direct consequence of `refactor` not changing the FM of L. The fact that L' has the same variants as L can be proven by checking that each product

p of L generates the same variant in L' and in L: this can be done by induction on the number of delta modules and delta operations used to generate the variant of p in L. ☐

Recall that the notion of increasing (resp. decreasing) monotonicity satisfied by the refactored SPL depends on the properties of the original SPL, as pointed out at the beginning of Section 5.1.1 (resp. Section 5.2.1).

Theorem 2 (Complexity)**.** *The space complexity of the* `refactor` *algorithms is: i) constant in the size of IFJ code; ii) linear in the number of delta operations; and iii) linear in the number of delta operations times the number of delta modules for the generation of the activation condition of the new delta modules. The time complexity of the* `refactor` *algorithms is quadratic in the number of delta operations.*

Proof (sketch). i) is a direct consequence of the algorithm not creating or duplicating IFJ code. ii) is more subtle: in the increasing refactor, $\mathtt{o}_1$ is replaced by one delta module containing one operation, and $\mathtt{o}_2$ is kept as it is; however in the decreasing refactor, to match all the **readds** and **removes** operations generated in `mergeToBase`, we need to consider that adding a class corresponds to one **adds** operation for the class name, and one **adds** operation for each of its fields. iii) it is straightforward to see that the length of the activation condition of the delta module created in function `mergeOperations` is linear in the number of delta modules in L. Finally, `refactor` is quadratic in time in the number of delta operations as it iterates over them with two inner loops (one in function `refactor`, one in function `manageOperation`). ☐

6 Related Work

To the best of our knowledge, refactoring in the context of DOP has been studied only in [16] and [7]. The former considers product lines of Java programs, while the latter considers delta modeling of software architectures. We refer to [16] for the related work in the FOP or annotative approaches. Note that both of these approaches are monotonic by construction (FOP being increasing, and annotative being decreasing), and so no refactoring algorithms to achieve monotonicity exist for them. In [16], a catalogue of refactoring and code smells is presented, and most of them focus on changing one delta module, one feature at a time. 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 **modifies** and **adds** operations. *Resolve Removal Action* eliminates **removes** operations also by changing the application condition of previous **modifies** and **adds** operations. Other refactoring algorithms focus on how to enable extractive SPL development for FOP [1, 12]. These works are related to ours, as DOP natively supports extractive SPL development: refactoring such a SPL into an increasing monotonic one using our algorithms is close to adapting this SPL to FOP.

7 Conclusion and Future Work

In this paper, we presented two refactoring algorithms with the goal of lowering the complexity of the input SPL, by removing opposite **adds** and **removes** operations. These algorithms work by removing one kind of operation from the input SPL, either **adds** or **removes**, and so they do not duplicate code nor change the structure of the input SPL, except for the parts related to the removed operation.

We plan four lines of future work for monotonicity in DOP. First, we would like to investigate alternative means to reach (a possibly more flexible version of) monotonicity. Second, complementarily to our algorithms, one could consider also refactoring code. For instance, splitting the definition of a method into several ones would help into transforming **modifies** operations in **voids** operations. Third, we would like to identify specific analysis scenarios where monotone product lines are simpler to analyze. Fourth, we plan to develop case studies in order to evaluate the advantages and the drawbacks of the proposed refactorings.

Acknowledgements. We are grateful to the FMSPLE 2016 anonymous reviewers for many comments and suggestions for improving the presentation.

References

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, New York, NY, USA, 2006. ACM. `doi:10.1145/1173706.1173737`.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. of ICSE 2003*, pages 187–197. IEEE, 2003. `doi:10.1109/TSE.2004.23`.

[3] L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Inf.*, 50(2):77–122, 2013. `doi:10.1007/s00236-012-0173-z`.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[5] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of GPCE 2005*, pages 422 – 437. 2005. `doi:10.1007/11561347_28`.

[6] M. Fowler. 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*, page 256, 2002. `doi:10.1007/3-540-45672-4_31`.

[7] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Evolving delta-oriented software product line architectures. *CoRR*, abs/1409.2311, 2014. `doi:10.1007/978-3-642-34059-8_10`.

[8] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001. `doi:10.1145/503502.503505`.

[9] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, July 2012. `doi:10.1145/2211616.2211617`.

[10] C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002. `doi:10.1109/MS.2002.1020284`.

[11] https://github.com/gzoumix/IFDJTS.

[12] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM. `doi:10.1145/1134285.1134303`.

[13] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. of ECOOP 2005*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005. `doi:10.1007/11531142_8`.

[14] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proc. of SPLC 2010*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010. `doi:10.1007/978-3-642-15579-6_6`.

[15] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Proc. of FOSD 2010*. ACM, 2010. `doi:10.1145/1868688.1868696`.

[16] S. Schulze, O. Richers, and I. Schaefer. Refactoring delta-oriented software product lines. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 73–84, New York, NY, USA, 2013. ACM. `doi:10.1145/2451436.2451446`.