Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking

Refactoring Delta-Oriented Product Lines to Enforce Guidelines for Efficient Type-checking*

Ferruccio Damiani and Michael Lienhardt

University of Torino, Italy
{ferruccio.damiani, michael.lienhardt}@unito.it

Abstract. A Software Product Line (SPL) is a family of similar programs generated from a common code base. Delta-Oriented Programming (DOP) is a flexible and modular approach to construct SPLs. Ensuring type safety in an SPL (i.e., ensuring that all its programs are well-typed) is a computationally expensive task. Recently, five guidelines to address the complexity of type checking delta-oriented SPLs have been proposed. This paper presents algorithms to refactor delta-oriented SPLs in order to follow the five guidelines. Complexity and correctness of the refactoring algorithms are stated.

1 Introduction

A *Software Product Line* (SPL) is a family of similar programs, called *variants*, with well documented variabilities [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* which 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. In DOP, variants are generated by selecting a valid set of features from the FM, which activates the corresponding delta modules that are then applied in order to the base program. 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

* 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.

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 transforming a set of existing programs into an SPL) SPL development [11].

Different type checking approaches for DOP have been studied [7,3,5] and some of them have been implemented for the ABS modeling language [10,1]. Recently [5], five programming guidelines that makes type checking more efficient have been proposed. Some of these guidelines are quite straightforward to follow, like the absence of useless operation (Section 4), others include subtleties and transforming an existing SPL into one that follows such guidelines can be quite challenging. This paper recalls the five guidelines and introduces for each of them an algorithm to refactor any delta-oriented SPL into an equivalent one that follows the given guideline. We illustrate these algorithms on a simple running example and discuss the complexity and correctness of the refactoring algorithms.

Section 2 introduces the example that will be used throughout the paper and recalls the Imperative Featherweight Delta Java (IFΔJ) core calculus for delta-oriented SPLs of Java programs. Section 3 introduces some auxiliary notations. Sections 4—8 present the five guidelines and their refactoring algorithms, respectively. Section 9 briefly discusses related work. Section 10 concludes the paper.

2 The Imperative Featherweight Delta Java Calculus

In this section, we introduce the structure of an IFΔJ SPL, the running example of this paper and briefly recall the IFΔJ [3] core calculus. In IFΔJ there is no concrete syntax for the feature model and the configuration knowledge of an SPL L. We instead use the following notations: $L.\texttt{features}$ is the set of features; $L.\texttt{products}$ specifies the products (i.e., a subset of the power set $2^{L.\texttt{features}}$); $L.\alpha$ maps each delta module name `d` to its activation condition; and $L.\texttt{order}$ (or $<_L$, for short) is the application ordering between the delta modules. Both the set of valid products and the activation condition of the delta modules are expressed as propositional logic formulas Φ where propositional variables are feature φ. A formula Φ represents the set of products $\{\overline{\varphi} \mid \overline{\varphi} \text{ validates } \Phi\}$ (see [2] for a discussion on other possible representations) and is described with the following syntax:

$$\Phi \ ::= \ \textbf{true} \ \mid \ \varphi \ \mid \ \Phi \Rightarrow \Phi \ \mid \ \neg\Phi \ \mid \ \Phi \wedge \Phi \ \mid \ \Phi \vee \Phi.$$

As usual, we say that a propositional formula Φ is *valid* if it is true for all values of its propositional variables. To avoid over-specification, the order $<_L$ can be partial. We assume *unambiguity* of the product line, i.e., for each product, any total ordering of the activated delta modules that respects $<_L$ generates the same variant (see [12,3] for a discussion on effective means to ensure unambiguity).

$EPL.\texttt{features} = \{\text{fLit}, \text{fAdd}, \text{fToInt}, \text{fToString}, \text{fEval1}, \text{fEval2}\}$
$EPL.\texttt{products} = \text{fLit} \wedge \text{fToInt} \wedge (\text{fEval1} \Rightarrow \text{fToString}) \wedge (\text{fEval1} \vee \text{fEval2}) \wedge \neg(\text{fEval1} \wedge \text{fEval2})$

$EPL.\texttt{order} = \{\texttt{dAdd}\} <_L \{\texttt{d\_notTostr}, \texttt{dAdd\_notTostr}\} <_L \{\texttt{dRMEval1}\} <_L \{\texttt{dEval2}\}$
$EPL.\alpha = \texttt{dAdd} \mapsto \text{fAdd},$
$\texttt{d\_notTostr} \mapsto (\neg\text{fToString}),\ \texttt{dAdd\_notTostr} \mapsto (\text{fAdd} \wedge \neg\text{fToString}),$
$\texttt{dRMEval1} \mapsto \neg\text{fEval1},\ \texttt{dEval2} \mapsto \text{fEval2}$

```
// Base program
class Exp extends Object { // To be used only as a type (i.e., not to be instantiated)
  Int toInt() { return new Int(); }
  String toString() { return ""; }
  Lit eval() { return (new Lit()).setLit(this.toInt()); }
}
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 Test extends Object {
  String test(Exp x) { return x.eval().toString(); }
}
// Delta Modules
delta dAdd {
  adds class Add extends Exp {
    Exp a;   Exp b;
    Int toInt() { return this.a.toInt().add(this.b.toInt()); }
    String toString() { return this.a.toString() + "+" + this.b.toString(); }
  }
}
delta d_notTostr {
  modifies class Exp { removes toString; }
  modifies class Lit { removes toString; }
}
delta dAdd_notTostr {  modifies class Add { removes toString; }  }
delta dRMEval1 {  modifies class Exp { removes eval }  }
delta dEval2 {  modifies class Exp { adds Int eval() {return this.toInt();} }  }
```

Fig. 1. Expression Product Line: FM (top), CK (middle), AB (bottom)

The running example of this paper is derived from the *Expression Product Line* (EPL) benchmark [13] (see also [3]) which encodes the following grammar for expressions over integers:

Exp ::= Lit | Add Lit ::= <non−negative−integers> Add ::= Exp "+" Exp

The EPL has 6 products, described by two feature sets: one concerned with data—fLit, fAdd—and one concerned with operations —fToInt, fToString, fEval1, fEval2. Features fLit and fToInt are mandatory, while the other features are optional with the two following constraints: exactly one between fEval1 and fEval2 must be selected; and fEval1 requires fToString. Each variant of the EPL contains a class `Exp` that represents an expression equipped with a subset of the following operations: `toInt` returns the value of the expression as an integer (an object of class `Int`); `toString` returns the expression as a `String`; and `eval` returns in some variants the value of the expression as a `Lit` (the subclass of `Exp` representing literals), and in the other variants the value of the expression as an `Int`. The definition of the EPL example is given in Figure 1. The partial order

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

---

L	$::= FM \quad CK \quad AB$	Product Line
AB	$::= P \quad \overline{\Delta}$	Artifact Base
Δ	$::=$ **delta** `d` { $\overline{CO}$ }	Delta Module
CO	$::=$ **adds** CD $\mid$ **removes** `C` $\mid$ **modifies** `C` [**extends** `C`′] { $\overline{AO}$ }	Class Operation
AO	$::=$ **adds** AD $\mid$ **removes** `a` $\mid$ **modifies** MD	Attribute Operation

Fig. 2. Syntax of IFJ (top) and of IFΔJ (bottom)

L.`order` is expressed as a total order on a partition of the set of delta modules. To make the example more readable, in the artifact base we use the JAVA syntax for operations on strings and sequential composition—encoding in IFΔJ syntax is straightforward (see [3] for examples). Note that, in the method `Test.test` (in the base program), the expression `x.eval()` has type `Lit` if feature fEval1 is selected (for this reason feature fEval1 requires feature fToString) and type `Int` otherwise.

In the following, we first introduce the IMPERATIVE FEATHERWEIGHT JAVA (IFJ) calculus, which is an imperative version of FEATHERWEIGHT JAVA [9], and then we introduce the constructs for variability on top of it. The abstract syntax of IFJ is presented in Figure 2 (top). Following [9], 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 method's formal parameter. A program P is a sequence of class declarations $\overline{CD}$. A class declaration **class** `C` **extends** `C`′ { $\overline{AD}$ } comprises the name `C` of the class, the name `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 assumed to be acyclic. Type system, operational semantics, and type soundness for IFJ are given in [3].

The abstract syntax of the IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ) language is given in Figure 2 (bottom). An IFΔJ program L comprises: a feature model FM, a configuration knowledge CK, and an artifact base AB. Recall that we do not consider a concrete syntax for FM and CK and use the notations L.`features`, L.`products`, $L.\alpha$, and L.`order` ($<_L$ for short) introduced above.

The artifact base comprises a possibly empty or incomplete IFJ program P, and a set of delta modules $\overline{\Delta}$.

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 operational semantics of IFΔJ variant generation is given in [3].

Definition 1 (Getter on *FM* and *CK*). *Let Φ be extended to include module names* `d` *as propositional variables. The formula* $L.\texttt{FMandCK} \triangleq L.\texttt{products} \wedge \bigwedge_{\texttt{d}}(\texttt{d} \Leftrightarrow L.\alpha(\texttt{d}))$ *specifies the products and binds each variable* `d` *to the activation condition of module* `d` *(i.e., it specifies which modules are activated for each product).*[1]

3 Auxiliary Notations

In this section we introduce some auxiliary notations that will be used in the definition of our different refactoring algorithms. The following notation unifies delta operations on classes and on attributes in a single model, in order to uniformly manage these two kinds of operations in our refactoring algorithms.

Notation 1. *A* reference*, written ρ, is either a class name* `C` *or a qualified attribute name* `C.a`*. 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, and iii)* D *is the data associated with this operations. The data corresponding to adding an element (a class or an attribute) is the element itself; the data corresponding to removing an element is empty; and the data corresponding to modifying a class (resp. a method) is the new* **extends** *statement (resp. the new version of the method). Given an ADO* `o`*, we denote its operator as* `o.`**dok***, its reference as* `o.`ρ*, and its data as* `o.`D*.*

This notation is illustrated 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.

[1] The last occurrence of `d` in $L.\texttt{FMandCK}$ is not used as a variable: it is used as argument of the map $L.\alpha$ to obtain the activation condition of module `d`.

Example 1. The delta module `dEval2` in Figure 1 that modifies classes `Exp` by adding a method `eval` to it, is modeled with one ADO:

(**adds**, Exp.eval, Lit eval() { **return** this.toInt(); })

The following notation introduces the *Family Class Signature Table* (`FCST`) that is used to retrieve important type information from an SPL L.

Notation 2. *An* Attribute Type T *is either a field type* $\mathtt{C}$ *or a method type* $(\overline{\mathtt{C}}) \to \mathtt{C}'$. *Given the data of an attribute* D, *we write* $\mathtt{type}(\mathrm{D})$ *the type of this data (e.g.,* $\mathtt{type}(\mathtt{C}\ \mathtt{m}(\mathtt{C}'\ \mathtt{x}))$ *is* $(\mathtt{C}') \to \mathtt{C}$*). Given an attribute reference* $\rho = \mathtt{C.a}$, $L.\mathtt{FCST}[\rho]$ *returns the mapping* $[T_i, \Phi_i]_{i \in I}$ *stating which are the possible types* T_i *of* $\mathtt{C.a}$ *in* L, *with* Φ_i *being the condition for that attribute to have that type. We write* $\mathrm{dom}(L.\mathtt{FCST})$ *the set of references* ρ *such that* $L.\mathtt{FCST}(\rho) \neq \emptyset$. *Given two class names* $\mathtt{C}_1$ *and* $\mathtt{C}_2$, $L.\mathtt{FCST}(\mathtt{C}_1 <: \mathtt{C}_2)$ *returns the condition* Φ *for which* $\mathtt{C}_2$ *is the super class of* $\mathtt{C}_1$ *in* L. *Finally,* $L.\mathtt{FCST}(\rho)$ *is defined similarly to* $L.\mathtt{FCST}[\rho]$, *except that it follows the inheritance relation (e.g., if* $\mathtt{C.a} \in \mathrm{dom}(L.\mathtt{FCST})$ *and* $L.\mathtt{FCST}(\mathtt{C}' <: \mathtt{C}) \neq \mathbf{false}$, *we have* $\mathtt{C}'.\mathtt{a} \in \mathrm{dom}(L.\mathtt{FCST})$*),*

Finally, our last notation is used to iterate over the delta modules of an SPL.

Notation 3. *The set of delta module names declared in* L *is denoted as* $\mathtt{dm}(L)$ *and we write* $L.\mathtt{d}$ *the delta module named* `d` *in* 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* $L.\mathtt{order}$.

4 G1: Absence of Useless Operations

A simple way to make typing SPL more efficient is to avoid cluttering it with code that will never be included into the SPL's variants. We call *useless operation* a delta operation that introduces some code or subtyping relation that will never be present in a variant, and avoiding such operation has two benefits: first, it makes typing quicker because part of the typing process is not lost on checking useless declarations; and second, it makes typing complete as all discovered typing errors correspond to an error in one of the SPL's variants. The guideline and the definition of the enforced property are presented as follows:

G1 Ensure that the product line does not contain useless operations.

Definition 2 (Useless operation and module). *The declaration, addition or modification of an element* ρ *in a module* `d` *is* useless *iff there exists a set of delta modules* $\{\mathtt{d}_i \mid 1 \leq i \leq n\}$ *that are applied after* `d` *and that either remove or replace (i.e., modifies it without calling* `original`*)* ρ *such that* $(L.\mathtt{FMandCK} \wedge \mathtt{d}) \Rightarrow \bigvee_{1 \leq i \leq n} \mathtt{d}_i$ *is valid. A module* `d` *is* useless *iff* $L.\mathtt{products} \Rightarrow \neg L.\alpha(\mathtt{d})$ *is valid.*

The avoidance and removal of useless operations is a classic concept in software product lines, and some refactoring algorithms and code smells have already been defined to address this problem. In particular, [16] proposes a large set of

code smells and refactoring algorithms for delta-oriented programming, which includes the smells *Dead Delta Action*, *Dead Delta Module*, and *Empty Delta Module* with corresponding refactoring algorithms. Applying these code smells to identify which operations and delta modules can safely be removed, and the corresponding refactoring algorithms to actually remove them would enforce G1 in any delta-oriented SPL.

5 G2: Type Uniformity

One of the causes of the complexity of type checking a delta-oriented SPL is the fact that two different modules may declare the same attribute with two different types. For instance, our EPL example contains two variations of the method eval: one that returns an Int and one that returns a Lit. Hence, for the type system to be complete, it needs to check the method Test.test twice, once for each possible type of the eval method. In general, the fact that a unique attribute can have several types may cause a method to be type-checked an exponential number of times.

Type uniformity is the property that all attributes declared in the SPL only have one type. Type checking is thus simpler (a method must be type-checked only once), and also using and extending such SPL is simpler.

The guideline and the definition of the enforced property are presented as follows:

G2 Ensure that the product line is type-uniform.

Definition 3 (Type-uniformity). *A product line L is* type-uniform *iff for all* $\texttt{C.a} \in \mathrm{dom}(L.\texttt{FCST})$*, the set* $L.\texttt{FCST}(\texttt{C.a})$ *is a singleton.*

Figure 3 presents the refactorG2 refactoring algorithm that transforms any IFΔJ SPL to enforce G2. It uses an auxiliary algorithm `refactorG2Data`, presented in Figure 4. The algorithm takes in input the SPL L to refactor and a *renaming function* σ. This function σ is injective, takes in input an attribute with its type and returns a new attribute name.

The algorithm is constructed with a loop iterating over all the delta operations in the SPL that renaming all encountered attributes. We distinguish two kind of operations: **removes** that does not associate data nor a type to the attribute it removes; while **adds** and **modifies** do associate data and a type to the manipulated attribute. The **removes** operation is handled in lines 4–17. An interesting property of the **removes** operation is that it can be applied on an attribute, whatever its type is. Consider for instance adding the operation $(\textbf{removes}, \texttt{Exp.eval}, \emptyset)$ to the EPL example: such operation can be applied to both variations of the eval method. Therefore, if the first and second variations of eval were respectively renamed eval1 and eval2, the remove operation must be duplicated so to still capture the two variations of the eval method. This duplication is handled as follows: line 5 identifies all the possible types of the attribute and either there is just one type, in which case no duplication is required and the

```
1  RefactorG2(L, σ) =
2    for d ∈ dm(L) do
3      for o ∈ L.d do
4        if (o.dok = removes) and (∃C, a, o.ρ = C.a) then // 1. the removes case
5          {T_i ↦ Φ_i}_{1≤i≤n} ←L.FCST[o.ρ]
6          if n = 1 then
7            o.ρ ←C.σ(T_1, a) // in place renaming
8          else
9            L.d ←L.d \ o // extract the operation from the module: we duplicate it
10           for i ∈ [1..n] do
11             L ←L + d' fresh with {
12               L(d') ←{ (o.dok, C.σ(T_i, a), ∅) }
13               L.α(d') ←L.α(d) ∧ Φ_i
14               L.order(d') ←L.order(d)
15             }
16           done
17         fi
18       else if ∃C, a, o.ρ = C.a then // 2. the adds and modifies case
19         o.ρ ←C.σ(type(o.D), a)
20         (D_i, Φ_i)_{1≤i≤n} ← refactorG2Data(L, σ, o.D)
21         if n = 1 then
22           o.D ←D_1 // in place renaming
23         else
24           L.d ←L.d \ o // extract the operation from the module: we duplicate it
25           for i ∈ [1..n] do
26             L ←L + d' fresh with {
27               L(d') ←{ (o.dok, o.ρ, D_i) }
28               L.α(d') ←L.α(d) ∧ Φ_i
29               L.order(d') ←Lorder(d)
30             }
31           done
32         fi
33       fi
34     done
35   done;
```

Fig. 3. The Type Uniformity Refactoring Algorithm

renaming is directly applied to the attribute (line 7); or there are several types, in which case the operation is extracted from its module (line 9), and duplicated in a fresh delta module, one new operation for each type of the attribute (the **for** loop in lines 10–16). When duplicating, the orders of the new delta modules are the same as before but the activation conditions are restricted to be active only for the considered variation of the attribute (line 13). The **adds** and **modifies** operations are handled in lines 18–32. Here, the type of the attribute is given in the data o.D so that renaming the attribute can be done directly (line 19). However, if the attribute is a method, its data (i.e., its body) can contain field accesses and method calls that must be renamed as well. Moreover, as the types of these field accesses and method calls are not given, we need to consider all the possible types of these attributes (similarly to the **removes** case), and duplication may occur. The renaming and duplication of data is done in the auxiliary algorithm `refactorG2Data` presented in Figure 4 and follows the same principle as the duplication done in the **removes** case. This `refactorG2Data` algorithm takes in input the SPL L (to have access to its type information L.FCST), the renaming function σ and the data to be renamed and duplicated. It returns a set

$$\textsc{E:Var}\ \frac{\Gamma(\mathtt{x}) = \mathtt{C}}{\Gamma, \sigma \vdash \mathtt{x} : [(\mathtt{x}, \mathtt{C}) \mapsto \mathbf{true}]} \qquad \textsc{E:Null}\ L, \Gamma, \sigma \vdash \mathbf{null} : [(\mathbf{null}, \bot) \mapsto \mathbf{true}]$$

$$\textsc{E:Access}\ \frac{L, \Gamma, \sigma \vdash e : [(e_i, T_i) \mapsto \Phi_i]_{i \in I} \quad I' = \{i \mid i \in I \wedge T_i.\mathtt{f} \in \mathrm{dom}(L.\mathtt{FCST})\} \quad \forall i \in I',\ L.\mathtt{FCST}(T_i.\mathtt{f}) = [\mathtt{C}_j \mapsto \Phi_j]_{j \in J_i}}{L, \Gamma, \sigma \vdash e.\mathtt{f} : [(e_i.\sigma(T_j, \mathtt{f}), T_j) \mapsto \Phi_i \wedge \Phi_j]_{i \in I', j \in J_i}}$$

$$\textsc{E:Call}\ \frac{\begin{array}{c} L, \Gamma, \sigma \vdash e : [(e_i, T_i) \mapsto \Phi_i]_{i \in I} \\ L, \Gamma, \sigma \vdash e_k : [(e_l, T_l) \mapsto \Phi_l]_{l \in L_k} \quad I' = \{i \mid i \in I \wedge T_i.\mathtt{m} \in \mathrm{dom}(L.\mathtt{FCST})\} \\ \forall i \in I',\ L.\mathtt{FCST}(T_i.\mathtt{m}) = [((\mathtt{C}_{1,j}, \ldots \mathtt{C}_{m_j,j}) \to \mathtt{C}_j) \mapsto \Phi_j]_{j \in J_i} \quad J'_i = \{j \mid j \in J_i \wedge m_j = n\} \end{array}}{L, \Gamma, \sigma \vdash e.\mathtt{m}(e_1, \ldots, e_n) : \left[\begin{array}{c} (e_i.\sigma((\mathtt{C}_{1,j}, \ldots \mathtt{C}_{m_j,j}) \to \mathtt{C}_j, \mathtt{m})(e_{l_1}, \ldots, e_{l_n}), \mathtt{C}_j) \\ \mapsto \Phi_i \wedge (\bigwedge\nolimits_{1 \le k \le n} \Phi_{l_k}) \wedge \Phi_j \end{array}\right]_{i \in I, j \in J'_i, l_k \in L_k}}$$

$$\textsc{E:New}\ L, \Gamma, \sigma \vdash \mathbf{new}\ \mathtt{C}() : [(\mathbf{new}\ \mathtt{C}(), \mathtt{C}) \mapsto \mathbf{true}] \qquad \textsc{E:Cast}\ \frac{L, \Gamma, \sigma \vdash e : [(e_i, T_i) \mapsto \Phi_i]_{i \in I}}{L, \Gamma, \sigma \vdash (\mathtt{C})e : [((\mathtt{C})e_i, \mathtt{C}) \mapsto \Phi_i]_{i \in I}}$$

$$\textsc{E:Assign}\ \frac{\begin{array}{c} L, \Gamma, \sigma \vdash e.\mathtt{f} : [(e_i, \mathtt{C}_i) \mapsto \Phi_i]_{i \in I} \\ L, \Gamma, \sigma \vdash e' : [(e_j, T_j) \mapsto \Phi_j]_{j \in J} \end{array}}{L, \Gamma, \sigma \vdash e.\mathtt{f} = e' : [(e_i = e_j, \mathtt{C}_i) \mapsto \Phi_i \wedge \Phi_j]_{i \in I, j \in J}} \qquad \textsc{R:Field}\ \mathtt{refactorG2Data}(L, \sigma, \mathtt{C}) = [\mathtt{C} \mapsto \mathbf{true}]$$

$$\textsc{R:Method}\ \frac{L, \overline{\mathtt{x} : \mathtt{C}}, \sigma \vdash e : [(e_i, T_i) \mapsto \Phi_i]_{i \in I}}{\mathtt{refactorG2Data}(L, \sigma, (\overline{\{\mathtt{C}\ \mathtt{x}\}}) \to \mathtt{C}'\ \{\ \mathbf{return}\ e;\ \}) = [(\overline{\{\mathtt{C}\ \mathtt{x}\}}) \to \mathtt{C}'\ \{\ \mathbf{return}\ e_i;\ \} \mapsto \Phi_i]_{i \in I}}$$

Fig. 4. Renaming and Duplicating Method Bodies

of renamed data D_i, together with the condition Φ_i in which this data is valid (similarly to the **removes** case, where the renaming $\sigma(T_i, \mathtt{a})$ was valid only when Φ_i was true). The computation of the duplicated data is done in line 20, and is handled as before: either there is just one data, in which case no duplication is required and the renaming is applied in place (line 22); or there are several data, in which case the operation is extracted from its module (line 24), and duplicated in a fresh delta module, one new operation for each data (the **for** loop in lines 25–31). When duplicating, the orders of the new delta modules are the same as before but the activation conditions are restricted to be active only for the considered variation of the data (line 28).

Application to the EPL example. As we previously discussed, our EPL example is not type uniform: the method `Exp.eval` may return an `Int` or a `Lit`. Applying our refactoring algorithm on this example, with the first (resp. second) variation of `eval` renamed in `eval1` (resp. `eval2`) has two effects on the EPL. First it simply changes the name of `eval` in the base program and in the delta modules `dRMEval1` and `dEval2` (line 19 of our algorithm). Second, because `Exp.eval` has two types, the duplication of the method `Test.test` raises two data, and so that method is extracted from its class and duplicated in two new delta modules, one where `eval1` is called while the second calls `eval2`. The following excerpt shows the modification done by our refactoring algorithm to the EPL example:

```
// Base program
```

```
class Exp extends Object { // To be used only as a type (i.e., not to be instantiated)
  ...
  Lit eval1() { return (new Lit()).setLit(this.toInt()); }
}
...
class Test extends Object { }
...
// Delta Modules
...
delta dRMEval1 {   modifies class Exp { removes eval1 }   }
delta dEval2 {   modifies class Exp { adds Int eval2() {return this.toInt();} }   }
delta dTestEval1 {   modifies class Test { adds String test(Exp x) { return x.eval1().toString(); } }   }
delta dTestEval2 {   modifies class Test { adds String test(Exp x) { return x.eval2().toString(); } }   }
```

Properties. Correctness and complexity of the refactoring algorithm are stated as follows.

- **Correctness of refactorG2.** Given a software product line L and an injective renaming function σ, for all product p of L such that the variant P of p can be generated, p is also a product of $\mathsf{RefactorG2}(L, \sigma)$, its variant P' can also be generated and we have that $P' = \sigma(P)$.
- **Complexity of refactorG2.** The refactoring algorithm RefactorG2 is exponential in time and space.

6 G3: Acyclic Inheritance

The acyclicity of the inheritance graph is an important property that must be enforced in most object-oriented programming languages such as Java. In general, checking this property is exponential in time (as it must be enforced for every variant of the SPL). However, this complexity can be reduced to linear if the global inheritance graph defined by the different **extends** statements present in the SPL is itself acyclic. Indeed, the inheritance graph of every variant is included into this global one so that the acyclicity of the global one implies the acyclicity of the inheritance graph of all the variants. It is thus natural to propose the following guideline:

G3 Ensure that the product line's inheritance graph is acyclic.

Definition 4. *Given a product line L, we write $\mathtt{C}_0 <: \mathtt{C}_1 <: \cdots <: \mathtt{C}_n \in L.\mathtt{FCST}$ iff for all $0 \leq i < n$, we have $L.\mathtt{FCST}(\mathtt{C}_i <: \mathtt{C}_{i+1}) \neq$ **false**. We say that the inheritance graph of a product line L is* acyclic *if there exists no sequence of classes $(\mathtt{C}_i)_{1 \leq i \leq n}$ such that $\mathtt{C}_0 <: \cdots <: \mathtt{C}_n <: \mathtt{C}_0 \in L.\mathtt{FCST}$ holds.*

To illustrate how our refactoring algorithm resolves inheritance loops, let us consider the example in Figure 5 which represents an inheritance loop between the classes $\mathtt{C}_i$ with $0 \leq i \leq 4$. Here, we suppose that the variants of the SPL are well-typed so that every variant includes just a part of this loop: in particular, the class $\mathtt{C}_0$ has a maximal ancestor, $\mathtt{C}_3$ in our example, meaning that there exists no variant in which $\mathtt{C}_4$ is an ancestor of $\mathtt{C}_0$. Therefore, it is correct to split (or duplicate) the classes $\mathtt{C}_1$, $\mathtt{C}_2$ and $\mathtt{C}_3$ in two: one copy that can be ancestors

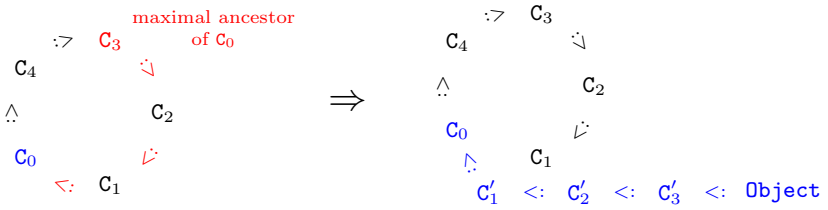

Fig. 5. Example of Inheritance Loop Resolution

```
1   refactorG3(L) =
2     for C0 <: ··· <: Cn <: C0 ∈ L.FCST do
3       Φ ←L.FCST(C0 <: C1)
4       i ←0
5       C ←C0
6       while(⊨ L.products ∧ Φ) and (i < n) do
7         C ← fresh
8         refactorG3Dup(L, Ci+1, C, Φ)
9         i ←i + 1;
10        Φ ←Φ ∧ L.FCST(Ci <: Ci+1)
11      done
12      if i < n then
13        L.FCST(C <: Object) ←L.FCST(C)
14      else
15        error
16      fi
17    done;
```

```
18  refactorG3Dup(L, C, C′, Φ) =
19    for d ∈ dm(L) do
20      for o ∈ L.d do
21        if (o.ρ = C or C ∈ o.D)
22            and (⊨ L.products ∧ L.α(d) ∧ Φ) then
23          L.d ←L.d \ o
24          L ←L + d1 fresh with {
25            L(d1) ←{ (o.dok, C′, o.D[C′/C]) }
26            L.α(d1) ←L.α(d) ∧ Φ
27            L.order(d1) ←Lorder(d)
28          } +d2 fresh with {
29            L(d2) ←{ (o.dok, C, o.D) }
30            L.α(d2) ←L.α(d) ∧ ¬Φ
31            L.order(d2) ←Lorder(d)
32          }
33        fi
34      done
35    done;
```

Fig. 6. The Acyclic Inheritance Refactoring Algorithm

of $\mathtt{C}_0$ and that do not extend $\mathtt{C}_4$, and one copy that inherit from $\mathtt{C}_0$. The result of this duplication on our example is shown on the right side of Figure 5.

Figure 6 presents our refactoring algorithm called refactorG3 that transforms any IFΔJ SPL into an equivalent that follows G3. Our algorithm takes in argument the SPL L to refactor, and iterates over all the inheritance loops to resolve them. The core of the algorithm is the **while** loop in lines 6–11 that iterates from the class $\mathtt{C}_0$ and duplicates all its ancestors until finding the maximal ancestor of $\mathtt{C}_0$. Our algorithm uses several variables. i is the index of the last class we considered in our algorithm; it is initialized to 0 (we start from the class $\mathtt{C}_0$) and incremented at each iteration of the while loop. Φ has two purposes but is principally the constraint that we use to find the maximal ancestor: it accumulates the activation condition of all the $\mathtt{C}_j <: \mathtt{C}_{j+1}$ statements, thus stating at each iteration of the **while** loop which products have the $\mathtt{C}_0 <: \cdots <: \mathtt{C}_i$ inheritance relation. Hence, when the formula $L.\mathtt{products} \wedge \Phi$ has a solution (written $\models L.\mathtt{products} \wedge \Phi$ in line 6), a variant with such inheritance relation exists, and the first time that formula does not have any solution, $\mathtt{C}_i$ is the maximal ancestor of $\mathtt{C}_0$. Finally, $\mathtt{C}$ is the name of the last duplicated class: it is initialized to $\mathtt{C}_0$, and at each iteration of the **while** loop, it is set to a fresh name, i.e., the name of the replica of the class $\mathtt{C}_{i+1}$. The duplication of the class $\mathtt{C}_{i+1}$ is done in the

auxiliary function refactorG3Dup, presented in the right part of Figure 6. The duplication iterates over all delta operations o in L, and each time it encounters a reference to the class to duplicate (either in o.ρ or in o.D), the operation o is duplicated: one variation keep referencing the original $\mathtt{C}_{i+1}$ name, but is activated only when Φ is false (i.e., when $\mathtt{C}_0$ may be an ancestor of $\mathtt{C}_{i+1}$), and one variation has its reference to $\mathtt{C}_{i+1}$ replaced by C and is activated only when Φ is true (i.e., when $\mathtt{C}_{i+1}$ may be an ancestor of $\mathtt{C}_0$). The conditional in lines 12–16 concludes the resolution of the loop: either we found the maximal ancestor of $\mathtt{C}_0$, in which case we set its super class to Object (in line 13), or there is no maximal ancestor, in which case there exists a product that contains the full loop: hence, L is ill-typed and our algorithm returns an error (in line 15).

Properties. Correctness and complexity of the refactoring algorithm are stated as follows.

- **Correctness of refactorG3.** Given a software product line L such that non of its variant contains an inheritance loop, for all product p of L such that the variant P of p can be generated, p is also a product of RefactorG3(L, σ), its variant P' can also be generated and we have that P' is identical to P, up to class renaming (due to the fresh class name creation in line 7).
- **Complexity of refactorG3.** The refactoring algorithm RefactorG3 is exponential in time and space.

7 G4: Non Overlapping Modules

Due to the possibly exponential duplication of code it introduces, refactoring an SPL into a type-uniform equivalent may not be advisable in some cases. The following guideline aims at helping the understanding of an SPL implementation by decoupling the sources of non type-uniformity.

G4 Ensure that, for all distinct modules $\mathtt{d}_1$ and $\mathtt{d}_2$, if the set comprising $\mathtt{d}_1$ and $\mathtt{d}_2$ is not type-uniform then their activation conditions are mutually exclusive.

Figure 7 presents our refactoring algorithm called refactorG4 that transforms any IFΔJ SPL into an equivalent that follows G4. This algorithm is based on the fact that G4 is a direct consequence of L not containing any remove operation on an element ρ being followed by an opposite **adds** operation. Therefore, our algorithm iterates over all the delta operations in the input SPL L and looks for an **adds** operation that follows a **removes** operation. The iteration is done in *descending order*: we use the operator $\downarrow_L$ that orders its input set following the opposite order of $<_L$. When a **adds** and an opposite and preceding **removes** operations are found (in lines 10–11), the algorithm calls the function refactorG4Ext which updates the activation condition of the **removes** operation and then calls the function refactorG4Upd which updates the activation condition of the **adds** and **modifies** operations that could occur before.

```
1  refactorG4(L) =
2    for d ∈↓_L(dm(L)) do
3      for o ∈ L.d do
4        if o.dok = adds then
5          Φ ←¬L.α(d)
6          for d′ ∈↓_L(before(d)) do
7            for o′ ∈ L.d′ do
8              if (o′.dok = removes)
9                  and (o′.ρ = o.ρ) then
10               L ←refactorG4Ext(L, d′, o′, Φ)
11               L ←refactorG4Upd(L, d′, o′.ρ, Φ)
12             fi
13           done
14         done
15       fi
16     done
17   done;

18 refactorG4Ext(L, d, o, Φ) =
19   L.d ←L.d \ o
20   L ←L + d′ fresh with {
21     L(d′) ←{ o}
22     L.α(d′) ←L.α(d) ∧ Φ
23     L.order(d′) ←L.order(d)
24   }
25   return L;
26
27 refactorG4Upd(L, d, ρ, Φ) =
28   for d′ ∈↓_L(before(d)) do
29     for o ∈ L.d′ do
30       if o.ρ = ρ then
31         if o.dok = removes then
32           return L
33         else
34           L ←refactorG4Ext(L, d′, o, Φ)
35         fi
36       fi
37     done
38   done
39   return L;
```

Fig. 7. Changing the Activation Condition of Operations to Achieve G4

Application to the EPL example. Our EPL example does not follow the G4 guideline: the **true** activation condition of the base program (which introduces the first variation of the `eval` method) has a non-empty intersection with the delta module `dEval2`. Applying our refactoring algorithm on this example has two effects on the EPL. First, the activation condition of `dRMEval1` is changed to ¬fEval1 ∧ ¬fEval2 due to the call to refactorG4Ext in line 10. Note that, following Definition 2, this delta module is now *useless* as it can never be activated anymore. Second, the method `Exp.eval` is extracted from the base program and put in a fresh delta module with ¬fEval2 as activation condition due to the call to refactorG4Upd in line 11.

Properties. Correctness and complexity of the refactoring algorithm are stated as follows.

- **Correctness of refactorG4.** Given a software product line L, the two following statements are equivalent:
  i) p is a product of L and P is its corresponding variant
  ii) p is a product of refactorG4(L) and P is its corresponding variant
- **Complexity of refactorG4.** The refactoring algorithm RefactorG4 is quadratic in time and space.

8 G5: Uniform Partitioning

We introduce our final guideline with the following consideration: implementing or modifying a product line involves editions of the feature model, the configuration knowledge and the artifact base that may affect only a subset of the

```
1  partitionG5(L) =
2    S_m ←∅
3    for (T, D) ∈ im(L.dFCST) do
4      if #D > 1 then
5        S_m ←S_m ∪ {D}
6      fi
7    done
8    res ←∅
9    for p ∈ L.products do
10     D ←∅
11     for d ∈ ⋃_{D'∈S_m} D' do
12       if p ⊨ L.α(d) then
13         D ←D ∪ {d}
14       fi
15     done
16     placed ←false
17     for (P, D') ∈ res do
18       if ∀D'' ∈ S_m, D ∩ D' ∩ D'' = ∅ then
19         placed ←true;
20         P ←P ∪ {p}
21         D' ←D' ∪ D
22         break;
23       fi
24     done
25     if ¬placed then
26       res ←res ∪ {({p}, D)}
27     fi
28   done
29   return res;
```

Fig. 8. Type-Uniform Partitioning of Products

products. For example, adding, removing or modifying a delta module `d` and its activation condition will affect only the products that activate `d`. Therefore, only the projection of the product line on the affected products needs to be re-analyzed. If such a projection is type-uniform, then a more efficient type-checking technique can be used (see [5] for more details). The following guideline naturally arises.

G5 i) Ensure that the set of products is partitioned in such a way that: each part S is type-uniform (i.e., the projection of the SPL on S is type uniform), and the union of any two distinct parts is not type-uniform.
ii) If the number of parts of such a partition is "too big", then merge some of them to obtain a "small enough" partition where only one part is not type-uniform.

Figure 8 presents our algorithm to compute a partition following the guideline G5.i, named partitionG5. To simplify the description of our algorithm, we will say in the rest of the section that two modules are in conflict iff they add the same attribute with two different types. The algorithm takes in input an SPL following G4 (so two conflicting delta modules cannot be activated by the same product) and is structured in two parts. First (in lines 2–7), we compute the sets of conflicting delta modules S_m by using an annex getter on L, L.`dFCST`. L.`dFCST` is an extention of L.`FCST` which maps every qualified attribute `C.a` declared in the SPL to a set of pairs (T, D) where T is a possible type of the attribute, and D is the set of delta modules that add the attribute `C.a` with the type T. The second part (in lines 8–28) compute the actual partition **res** tagged with annex information used during the computation: basically, **res** is a set of pairs (P, D) where P is the set of products in the partition, and D is the set of modules conflicting with other modules used by the products in P. The computation of **res** is done by iterating over all the products p of the SPL (written $p \in L$.`products` in line 9). First (in lines 10–15), the algorithm computes the set D of possibly conflicting modules activated by p (we write $p \vDash \Phi$ when p validates the formula Φ). Then (in lines 16–24), we try to add p in existing

partitions (P, D') by iterating over them and checking if a module in D is in conflict with a module in D'. If all existing partitions are in conflict with p, we add to **res** a new partition containing only p (in lines 25–27).

Additionally, we propose in Figure 9 an algorithm that performs an incremental type-check. As mentioned in the beginning of this section, it is common that well-typed software product lines are modified, and to ensure the well-typedness of the result, it is necessary to only verify a small part of the SPL. The algorithm in Figure 9, named typeG5, takes in input the originally well typed SPL L, the set of the added modules' names D, and the set of deleted modules U (modified modules are considered removed and added). The principle of this algorithm is to extract from L a much smaller SPL L' that contains all the necessary data to ensure that the well-typedness of P' is equivalent to the well-typedness of L. The construction of L' is done in three steps. First (in line 2), the auxiliary function typeG5Init creates L' with the same feature model as L, and all the modules in D. Moreover, it computes the set of elements E that are used or manipulated (i.e., added, removed or modified) by the modules in D: all delta operations manipulating these elements must be added to L' to type-check correctly the modules in D. To do so, it uses the function `els` that extracts the set of used or manipulated names from a delta operation `o`. Finally, it also computes the elements E' that were manipulated by the deleted modules in U: all delta operations manipulating or using these elements must be added to L' as they may not be well-typed anymore. The addition in L' of the delta operations manipulating or using the elements in E' is done in lines 3–14 by iterating over all the delta modules in L (except those in D that are already included in L'). In addition, this part of the algorithm extends E with the dependencies of the added delta operations: like before, all delta operations manipulating these elements must be added to L' to correctly type-check the added operations. The auxiliary function typeG5AddManip efficiently integrates E into L': as stated, this function adds to L' all the delta operations that manipulate the elements in E but also replaces all the method bodies with "**return null**": that way, there is no need to also add the dependencies of these operations in L'. Finally, we can type-check L' in line 16 to see if L is well-typed. Note that this algorithm does not require for the input SPL to follow the G4 guideline.

Application to the EPL example. As mentioned in Section 2, the EPL example has 6 products. Moreover, if we consider the variation of this example that follows G4 (discussed in Section7), there are two conflicting modules: `dEval2` and the one created by the refactorG4 algorithm. Therefore, the partition computed by our algorithm, which is unique in this case, has two elements P_1 and P_2 constructed as follows:

$$P_1 \triangleq \left\{ \begin{array}{c} \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval1},\mathsf{fToString}\} \\ \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval1},\mathsf{fToString},\mathsf{fAdd}\} \end{array} \right\} \quad P_2 \triangleq \left\{ \begin{array}{c} \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval2}\} \\ \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval2},\mathsf{fToString}\} \\ \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval2},\mathsf{fAdd}\} \\ \{\mathsf{fLit},\mathsf{fToInt},\mathsf{fEval2},\mathsf{fToString},\mathsf{fAdd}\} \end{array} \right\}$$

```
1  typeG5(L, D, U) =
2    (L′, E, E′) ←typeG5Init(L, D, U)
3    for d ∈ dm(L) \ D do
4      for o ∈ L.d do
5        if els(o) ∩ E′ ≠ ∅ then
6          E ←E ∪ els(o)
7          L′ ←L′ + d′ fresh with {
8            L′(d′) ←{ o }
9            L′.α(d′) ←L.α(d)
10           L′.order(d′) ←Lorder(d)
11         }
12       fi
13     done
14   done
15   L′ ←typeG5AddManip(L, E, L′)
16   type_check(L′);
17
18 typeG5Init(L, D, U) =
19   L′ ←L.products
20   E ←∅
21   for d ∈ D do
22     L′ ←L′ ∪ L.d
23     E ←E ∪ {els(o) | o ∈ L.d}
24   done
25   E′ ←∅
26   for Δ ∈ U do
27     E′ ←E′ ∪ {o.ρ | o ∈ Δ}
28   done
29   return (L′, E, E′);
30 typeG5AddManip(L, E, L′) =
31   for d ∈ dm(L) do
32     for o ∈ L.d do
33       if o.ρ ∈ E then
34         L′ ←L′ + d′ fresh with {
35           L′(d′) ←{ (o.dok, o.ρ, typeG5Simple(o.D)) }
36           L′.α(d′) ←L.α(d)
37           L′.order(d′) ←Lorder(d)
38         }
39       fi
40     done
41   done
42   return L′;
43
44   typeG5Simple(D) =
45   if D = (C̄{ x }) → C′ { return e; } then
46     return D[null/e]
47   else if D = class C extends C′ { ... } then
48     return D[null/e]
49   else
50     return D
51   fi;
```

Fig. 9. Efficient Typing of Modified Delta Modules

To illustrate our typeG5 algorithm, let consider that a programmer modifies the EPL example by removing the `dRMEval1` module (thus causing an error in the generation of all the variant with the feature fEval2 activated). Our algorithm, with the typeG5Init function, first initializes L' to contain only the feature model of the EPL example, E is empty and $E' = \{\texttt{Exp.eval}\}$. The second part of the algorithm adds to L' the method `Exp.eval` declared in the base program of EPL and the delta module `dEval2`. As the method `Exp.eval` depends on the class `Lit` and the methods `Lit.setLit` and `Exp.toInt`, these elements are added to L' by the function typeG5AddManip. The resulting SPL L' has the following form:

```
// Base program
class Exp extends Object { // To be used only as a type (i.e., not to be instantiated)
  Int toInt() { return new Int(); }
  Lit eval() { return (new Lit()).setLit(this.toInt()); }
}
class Lit extends Exp {
  Lit setLit(Int x) { return null; }
}
delta dEval2 {   modifies class Exp { adds Int eval() {return this.toInt();} }   }
```

Finally, our algorithm tries to type-check L' and fails as `dEval2` redefines `Exp.eval` on top of its original definition in the base program: the typing error in the modified EPL is thus correctly detected by simply looking at the smaller SPL L'.

Properties. The properties of the two algorithms are stated as follows.

- **Correctness of partitionG5.** Given a software product line L and $\{(P_i, D_i) \mid i \in I\} = \mathsf{partitionG5}(L)$, the partition of products $\{P_i \mid i \in I\}$ follows G5.i.
- **Complexity of partitionG5**. The algorithm partitionG5 is exponential in time and space.
- **Correctness and Completeness of typeG5.** Given a well-typed software product line L and another product line L', a set of delta module names D and a set of delta modules U such that L' is the result of removing the delta modules U from L, and adding to it the delta modules in D. Then, the two following properties are equivalent:
  - L' is well-typed
  - $\mathsf{typeG5}(L', D, U)$ is well-typed
- **Complexity of typeG5.** The algorithm partitionG5 is linear in time and space.

9 Related Work

To the best of our knowledge, refactoring in the context of DOP has been studied only in [16], [8] and [6]. We refer to [16] for the related work in the FOP or annotative approaches.

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. In particular, the code smells and corresponding refactoring algorithms *Dead Delta Action*, *Dead Delta Module*, and *Empty Delta Module* presented in this paper can be combined to achieve the guideline G1, as described in Section 4. In [8], similar refactoring primitives are considered, for delta-oriented programming over components.

In [6] we have presented two refactoring algorithms that, like the algorithm presented in this paper, change the full structure of a given SPL to enforce some property. The properties considered in [6] are *increasing-* and *decreasing-monotonicity*, which focus on which delta operations are used in the SPL. The main intended use of these fully automated algorithms is to transform an SPL into an auxiliary variation more suited for static analysis, such as type checking. In fact, these algorithms might introduce major changes to the structure of an SPL leading to a result that might confuse the developers.

10 Conclusion and Future Work

In this paper, we presented refactoring algorithms for delta-oriented SPLs to enforce the guidelines introduced in [5] and have stated their main properties.

In future work, we would like to prove the properties that we have stated, to implement the algorithms, and to develop case studies to evaluate them. Moreover, these algorithms can be straightforwardly adapted in two ways that could be useful in practice. First, they can be made user-driven, so a programmer could use them to change his working copy of the SPL. Second, they can be transformed into simple analysis tools which could identify where some properties (such as uniformity or inheritance acyclicity) are invalidated in an SPL.

References

1. https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta.
2. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. of SPLC 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
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. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In *Proc. of iFM*, volume 9681 of *LNCS*, pages 47–62. Springer, 2016. doi:10.1007/978-3-319-33693-0_4.
6. F. Damiani and M. Lienhardt. Refactoring delta-oriented product lines to achieve monotonicity. In *Proc. of FMSPLE*, volume abs/1604.00346 of *CoRR*, 2016. doi:10.4204/EPTCS.206.2.
7. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *Proc. of ISoLA (1)*, volume 7609 of *LNCS*, pages 193–207. Springer, 2012. doi:10.1007/978-3-642-34026-0_15.
8. 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.
9. 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.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012. doi:10.1007/978-3-642-25271-6_8.
11. C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002. doi:10.1109/MS.2002.1020284.
12. M. Lienhardt and D. Clarke. Conflict detection in delta-oriented programming. In *ISoLA*, pages 178–192, 2012. doi:10.1007/978-3-642-34026-0_14.
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.