On Type Checking Delta-Oriented Product Lines

On Type Checking Delta-Oriented Product Lines[⋆]

Ferruccio Damiani and Michael Lienhardt

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

Abstract. A Software Product Line (SPL) is a set of similar programs generated from a common code base. Delta Oriented Programming (DOP) is a flexible approach to implement SPLs. Efficiently type checking an SPL (i.e., checking that all its programs are well-typed) is challenging. This paper proposes a novel type checking approach for DOP. Intrinsic complexity of SPL type checking is addressed by providing early detection of type errors and by reducing type checking to satisfiability of a propositional formula. The approach is tunable to exploit automatically checkable DOP guidelines for making an SPL more comprehensible and type checking more efficient. The approach and guidelines are formalized by means of a core calculus for DOP of product lines of Java programs.

1 Introduction

A *Software Product Line* (SPL) is a set of similar programs, called *variants*, with a common code base and well documented variability [6]. *Delta-Oriented Programming* (DOP) [18, 19, 5] is a flexible transformational approach to implement SPLs. A DOP product line is described by 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 feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*. The AB provides language dependent code artifacts that are used to build the variants: it consists of a *base program* (that might be empty or incomplete) and of a set of *delta modules*, which are containers of modifications to a program (e.g., for Java programs, a delta module can add, remove or modify classes and interfaces). The CK connects the code artifacts in the AB with the features in the FM (thus defining a mapping from products to variants): it associates to each delta module an *activation condition* over the features and specifies an *application ordering* between delta modules [19]. DOP supports the automatic generation of variants based on a selection of features:

[⋆] 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.

once a user selects a product, the corresponding variant is derived by applying the delta modules with a satisfied activation condition to the base program according to the application ordering.

DOP is a generalization of *Feature-Oriented Programming* (FOP) [4, 22, 9]: the artifact base 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. Hence FOP product line development always start from base feature modules corresponding to mandatory features. Instead, DOP allows to use arbitrary code as a base program. For example, the base program can be empty and different variants can be used as base delta modules with pairwise disjoint activation conditions [20]. Therefore, DOP supports both proactive SPL development (i.e., planning all products/variants in advance) and extractive SPL development [15] (i.e., starting from existing programs). Moreover (see, e.g., [5]), the decoupling between features and delta modules allows to counter the optional feature problem [13], where additional glue code is needed in order to make optional features to cooperate properly. Due to the additional flexibility, in DOP it is more challenging than in FOP to efficiently type check a product line [5]. Type checking approaches for DOP have already been studied [8, 5], and implemented [1] for the ABS modeling language [12]. Although these approaches do not require to generate any variant, they involve an explicit iteration over the set of products, which becomes an issue when the number of products is large (a product line with n features can have up to 2^n products).

In this paper we propose a novel type checking approach for DOP by building on ideas proposed for FOP [22, 9]. Our approach represents an achievement over previous type checking approaches for DOP [5, 8] since it provides earlier detection of some type errors and does not require to iterate over the set of products. Like the techniques in [22, 9], our approach requires to check the validity of a propositional formula (which is a co-NP-complete problem) and can take advantages of the many heuristics implemented in SAT solvers (a SAT solver can be used to check whether a propositional formula is valid by checking whether its negation is unsatisfiable)—[22, 9] report that the performance of using SAT solvers to verify the propositional formulas for four non-trivial product lines was encouraging and that, for the largest product line, applying the approach was even faster than generating and compiling a single product. Moreover, our approach is designed to be tunable to take advantage of automatically checkable DOP guidelines that make a product line more comprehensible and type checking more efficient. We formalize the approach and guidelines by means of Imperative Featherweight Delta Java (IFΔJ) [5], a core calculus for DOP product lines where variants are written in an imperative version of Featherweight Java (FJ) [11].

Section 2 introduces an example that will be used through the paper and recalls IFΔJ. Section 3 introduces two DOP guidelines (*no-useless-operations* and *type-uniformity*). Section 4 gives a version of the approach tuned to exploit type-uniformity. Section 5 outlines a version that exploits no guidelines. Section 6 proposes other guidelines. Section 7 discusses related work. Section 8 concludes the paper by outlining planned future work. Proofs of the main results and a prototypical implementation are available in [2] (currently only the version of the approach in Section 4 is supported).

2 Model

In this section we introduce the running example of this paper and briefly recall the IFΔJ [5] core calculus. A product line L consist of a feature model, a configuration knowledge, and an artifact base. In IFΔJ there is no concrete syntax for the feature model and the configuration knowledge. We use the following notations: 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. 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 names φ (see [3] for a discussion on other possible representations):

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

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. We refer to [16, 5] for a discussion on an effective means to ensure unambiguity.

The running example of this paper is a version of the *Expression Product Line* (EPL) benchmark [17] (see also [5]) defined by the following grammar which describes a language of numerical expressions:

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

Each variant of the EPL contains a class `Exp` that represents an expression equipped with a subset of the following operations: `toInt`, which returns the value of the expression as an integer (an object of class `Int`); `toString`, which returns the expression as a `String`; and `eval`, which in some variants returns the value of the expression as a `Lit` (the subclass of `Exp` representing literals) and in the other variants returns it as an `Int`. 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. The other features are optional with the two following constraints: exactly one between fEval1 and fEval2 must be selected; and fEval1 requires fToString. The EPL is illustrated in Figure 1. The partial order 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 [5] for examples). Note that, in the method `Test.test` (in the base program), the

$$
\begin{array}{lcl}
EPL.\mathtt{features} & = & \{\mathsf{fLit}, \mathsf{fAdd}, \mathsf{fToInt}, \mathsf{fToString}, \mathsf{fEval1}, \mathsf{fEval2}\} \\
EPL.\mathtt{products} & = & \mathsf{fLit} \wedge \mathsf{fToInt} \wedge (\mathsf{fEval1} \Rightarrow \mathsf{fToString}) \wedge (\mathsf{fEval1} \vee \mathsf{fEval2}) \wedge \neg(\mathsf{fEval1} \wedge \mathsf{fEval2})
\end{array}
$$

$$
\begin{array}{lcl}
EPL.\mathtt{order} & = & \{\mathtt{dAdd}\} <_L \{\mathtt{d\_notTostr}, \mathtt{dAdd\_notTostr}\} <_L \{\mathtt{dEval1}, \mathtt{dEval2}\} \\
EPL.\mathtt{activation} & = & \mathtt{dAdd} \mapsto \mathsf{fAdd}, \\
 & & \mathtt{d\_notTostr} \mapsto (\neg\mathsf{fToString}),\ \ \mathtt{dAdd\_notTostr} \mapsto (\mathsf{fAdd} \wedge \neg\mathsf{fToString}), \\
 & & \mathtt{dEval1} \mapsto \mathsf{fEval1},\ \ \mathtt{dEval2} \mapsto \mathsf{fEval1}
\end{array}
$$

```
// 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 ""; }
}
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 dEval1 {   modifies class Exp { adds Lit eval() {return (new Lit()).setLit(this.toInt());} }   }
delta dEval2 {   modifies class Exp { adds Int eval() {return this.toInt();} }   }
```

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

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 IFJ calculus, which is an imperative version of FJ [11], and then we introduce the constructs for variability on top of it. The abstract syntax of IFJ is presented in Figure 2 (top). Following [11], 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

$$
\begin{array}{lll}
P & ::= \overline{CD} & \text{Program} \\
CD & ::= \textsf{\textbf{class}}\ \mathtt{C}\ \textsf{\textbf{extends}}\ \mathtt{C}\ \{\ \overline{AD}\ \} & \text{Class} \\
AD & ::= FD \ \mid\ MD & \text{Attribute (Field or Method)} \\
FD & ::= \mathtt{C}\ \mathtt{f} & \text{Field} \\
MD & ::= \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\ \{\textsf{\textbf{return}}\ e;\} & \text{Method} \\
e & ::= \mathtt{x} \ \mid\ e.\mathtt{f} \ \mid\ e.\mathtt{m}(\overline{e}) \ \mid\ \textsf{\textbf{new}}\ \mathtt{C}() \ \mid\ (\mathtt{C})e \ \mid\ e.\mathtt{f} = e \ \mid\ \textsf{\textbf{null}} & \text{Expression} \\
\hline
L & ::= FM\quad CK\quad AB & \text{Product Line} \\
AB & ::= P\quad \overline{\mathit{\Delta}} & \text{Artifact Base} \\
\mathit{\Delta} & ::= \textsf{\textbf{delta}}\ \mathtt{d}\ \{\ \overline{CO}\ \} & \text{Delta Module} \\
CO & ::= \textsf{\textbf{adds}}\ CD \ \mid\ \textsf{\textbf{removes}}\ \mathtt{C} \ \mid\ \textsf{\textbf{modifies}}\ \mathtt{C}\ [\textsf{\textbf{extends}}\ \mathtt{C}']\ \{\ \overline{AO}\ \} & \text{Class Operation} \\
AO & ::= \textsf{\textbf{adds}}\ AD \ \mid\ \textsf{\textbf{removes}}\ \mathtt{a} \ \mid\ \textsf{\textbf{modifies}}\ MD & \text{Attribute Operation}
\end{array}
$$

Fig. 2. Syntax of IFJ (top) and of IF$\mathit{\Delta}$J (bottom)

(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 [5].

The abstract syntax of the language IF$\mathit{\Delta}$J is given in Figure 2 (bottom). An IF$\mathit{\Delta}$J 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.`activation`, 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 $\overline{\mathit{\Delta}}$.

A delta module declaration $\mathit{\Delta}$ 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 class operations in a delta module must act on distinct classes, and the attribute operations in a class operation must act on distinct attributes. The operational semantics of IF$\mathit{\Delta}$J variant generation is given in [5].

We conclude this section with some notations and definitions. First, in the rest of the document, we will use the term *module* to refer to the base program or a delta module: we denote with `p` the name of the base program, and extend L.`activation` by convention, stating that L.`activation`(`p`) = **true**. Second, the *projection* of a product line on a subset S of its products is the product line obtained by restricting the L.`products` formula to describe only the products in S and by ignoring delta modules that are never activated. Third, the following definitions introduce auxiliary structures and getters that are useful to type check an IF$\mathit{\Delta}$J product line.

Definition 1 (FCST). *A* Class Signature *(CS) is a class declaration deprived of the bodies of its methods, it comprises the name of the class and of its superclass, and a mapping from attribute names to types. A* Family Class Signature *(FCS) is a more liberal version of class signature that may extend multiple classes and associate more than one type to each attribute name. A* Family Class Signature table *(FCST) is a mapping that associates to each class name* `C` *an FCS for* `C`*. The subtyping relation* $<:$ *described by an FCST can be cyclic. A* Class Signature Table *(CST) is a FCST that contains only class signatures and has an acyclic subtyping relation.*

To simplify the notation, except when stated otherwise, we always assume in the following a fixed product line $L = FM\ CK\ AB$. The FCST of L, denoted by $L.\mathtt{FCST}$, contains for each class `C` declared in AB all superclasses of `C` and all types of all attributes of `C`. Note that the FCST of L is defined only in terms of AB and it can be computed by a straightforward inspection of it. The FCST of a set of IFJ programs (or of a subset of AB) is defined similarly.

Definition 2 (Getters on AB). `add(C)` *is the set of modules that* add *the class* `C`*;* `remove(C)` *is the set of modules that* remove *the class* `C`*;* `modifyWEC(C)` *is the set of modules that modify the class* `C` without changing its **extends** clause*;* `modifyAEC(C)` *is the set of modules that modify the class* `C` also by changing its **extends** clause*;* `modify(C)` *is* `modifyWEC(C)`$\cup$`modifyAEC(C)`*;* `add(C.a)` *is the set of modules that* add *the attribute* `C.a`*;* `remove(C.a)` *is the set of modules that* remove *the attribute* `C.a`*;* `modify(C.a)` *is the set of modules that* modify *the attribute* `C.a`*;* `replace(C.m`*) is the set of modules that modify the method* `C.m` *without using calls to* `original` *(i.e.,* replace *its body); and* `wrap(C.m)` *is the set of modules that modify the method* `C.m` *by also using calls to* `original` *(i.e.,* wrap *its body).*

Definition 3 (Getter on FM and CK). *Let Φ be extended to include module names* `d` *as propositional variables. The formula* $L.\mathtt{FMandCK} \triangleq L.\mathtt{products} \wedge \bigwedge_{\mathtt{d}}(\mathtt{d} \Leftrightarrow L.\mathtt{activation}(\mathtt{d}))$ *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 Two Delta-Oriented Programming Guidelines

The first guideline is to avoid *useless operations*, i.e., declarations in P and **adds** or **modifies** in $\overline{\Delta}$ that introduce code that is never present in any of the variants.

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

For instance, in the product line obtained by projecting the EPL on the the products described by ¬fToString, the declarations of the methods with name toString in the base program and in the **adds** class operation in the delta module dAdd are useless. The notion of useless operation is formalized as follows (thus making Guideline G1 automatically checkable).

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

Definition 4 (Useless operation and module). *The declaration, addition or modification of an attribute* $\mathtt{C.a}$ *in a module* $\mathtt{d}$ *is* useless *iff the formula* $(L.\mathtt{FMandCK} \wedge \mathtt{d}) \Rightarrow \bigvee_{\mathtt{d}'} \mathtt{d}'$ *(with* $\mathtt{d}' \in \mathtt{remove(C.a)} \cup \mathtt{remove(C)} \cup \mathtt{replace(C.a)}$ *and* $(\mathtt{d} <_L \mathtt{d}')$*) is valid. An* **extends** *clause introduced in a class* $\mathtt{C}$ *by a module* $\mathtt{d}$ *is* useless *iff the formula* $(L.\mathtt{FMandCK} \wedge \mathtt{d}) \Rightarrow \bigvee_{\mathtt{d}'} \mathtt{d}'$ *(with* $\mathtt{d}' \in \mathtt{remove(C)} \cup \mathtt{modifyAEC(C)}$ *and* $(\mathtt{d} <_L \mathtt{d}')$*) is valid. A module* $\mathtt{d}$ *is* useless *iff* $L.\mathtt{products} \Rightarrow \neg L.\mathtt{activation(d)}$ *is valid.*

The second guideline is to have consistent declarations over the whole SPL (the FOP case-studies presented in [22] adhere to this guideline). For IFΔJ (since IFJ has no method overloading and field shadowing), this means that two declarations of the same attribute (of the same class) in two different modules must have the same type.[2] We call this property *type-uniformity.* It can be straightforwardly formalized by exploiting the family class signature table of the product line.

Definition 5 (Type-uniformity). *A FCST* $\mathtt{FCST}$ *is* type-uniform *iff:*

- $\forall \mathtt{C} \in \mathrm{dom}(\mathtt{FCST})$, $\forall \mathtt{a} \in \mathrm{dom}(\mathtt{FCST(C)})$ *the set* $\mathtt{FCST(C.a)}$ *is a singleton; and*
- $\forall \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3 \in \mathrm{dom}(\mathtt{FCST})$ *such that* $\mathtt{C}_1 <: \mathtt{C}_2$ *and* $\mathtt{C}_1 <: \mathtt{C}_3$*, we have:* $\forall \mathtt{a} \in \mathrm{dom}(\mathtt{FCST}(\mathtt{C}_2)) \cap \mathrm{dom}(\mathtt{FCST}(\mathtt{C}_3))$, $\mathtt{FCST}(\mathtt{C}_2.\mathtt{a}) = \mathtt{FCST}(\mathtt{C}_3.\mathtt{a})$

An IFΔJ product line (or a subset of its artifact base, or a set of IFJ programs) is type-uniform *iff its FCST is type-uniform.*

Our second guideline is thus stated as follows (and it can automatically be checked by a straightforward inspection of the FCST).

G2 Ensure that the product line is type-uniform.

The EPL is not type-uniform, because of the method `eval` of class `Exp`, that is added with two different types by delta modules `dEval1` and `dEval2`, respectively. Instead, both its two projections respectively described by the mutually exclusive features fEval1 and fEval2 are type-uniform.

We say that an IFΔJ product line is *variant-type-uniform* to mean that: (i) its variants can be generated; and (ii) the FCST of the set of its variants is type-uniform. The following proposition illustrate how type-uniformity relates to variant-type-uniformity.

Proposition 1. *Let L be an IFΔJ product line such that its variants can be generated. If L is type-uniform, then it is variant-type-uniform. If L satisfies Guideline G1 and is variant-type-uniform, then it is type-uniform.*

4 Type Checking for Type-Uniform IFΔJ

This section presents a version of the type checking approach tuned to exploit Guideline G2 and states its correctness and completeness. Type-uniformity makes type checking more efficient. The approach is modularized in three independent parts: *partial typing*, *applicability*, and *dependency.* All the parts rely on the FCST of the product line (see Definition 1).

[2] Note that, since the type system of IFJ is nominal, a class may have different sets of attributes in different variants.

Product Line Partial Typing Partial typing checks that all fields, methods and classes in AB type-check with respect to the product line FCST (i.e., with respect to declarations made in AB). Partial typing does not use any knowledge about valid feature combinations (it does not use FM and CK), so it does not guarantee that variants are well-typed, as delta modules may be activated or not. However, it guarantees that variants that have their inner dependencies satisfied (i.e., all used classes, methods and fields are declared) are well-typed.

The IFΔJ partial-type-system is a straightforward extension of the (standard) IFJ type system [5] that: (i) includes rules for the new syntactic constructs of IFΔJ; (ii) checks well-typedness with respect to the product line FCST (instead of the program CST); and (iii) allows to introduce a same class or attribute in different modules of AB (e.g., a class of name C may be added by different delta modules).

The projection of the EPL described by feature fEval1 is type-uniform. Its artifact base (which is obtained from the EPL artifact base in Fig. 1 by dropping the delta module dEval2) is accepted by partial typing, even if the method Exp.eval might not be available in some variant (in principle the delta module dEval1 might not be selected). This is because the way the method Exp.eval is used in the method Test.test in the base program is correct with respect to its definition in the delta module dEval1 (it takes no parameters and returns a Lit object).

Product Line Applicability Applicability ensures that variants can actually be generated (variant generation fails if, e.g., a delta module that adds a class C is applied to an intermediate variant that already contains a class named C). It is formalized by a constraint ensuring that, during variant generation, each delta operation is applied to an intermediate variant on which that operation is defined. For instance, for adding a class C, this class must not be present in the intermediate variant (either it never was added, or it was removed at some point). The applicability constraint comprises three validation parts: element addition (either a class or an attribute), element removal, and element modification.

In the following we use ρ to denote either a class name C or a fully qualified attribute name C.a. The constraint for checking that an element ρ can be added is as follows:

$$\mathtt{appADD}(\rho) \triangleq \bigwedge_{\mathtt{d}\neq\mathtt{d}'} \mathtt{d} \wedge \mathtt{d}' \Rightarrow \bigvee_{\mathtt{d}''} \mathtt{d}'' \quad \text{with} \begin{cases} \mathtt{d}, \mathtt{d}' \in \mathtt{add}(\rho),\ \mathtt{d}'' \in \mathtt{remove}(\rho) \\ \text{and } \mathtt{d} <_L \mathtt{d}'' <_L \mathtt{d}' \end{cases}$$

It ensures that all **adds** operations are performed on a partial variant that does not contain the added element: basically, it requires that if two delta modules $\mathtt{d}$ and $\mathtt{d}'$ add the same element, then there must be another delta module $\mathtt{d}''$ in between that removes it.

The constraint for removal of an element ρ is slightly more complex:

$$\mathtt{appRM}(\rho) \triangleq \bigwedge_{\mathtt{d}} \mathtt{d} \Rightarrow (\bigvee_{\mathtt{d}_1} \mathtt{d}_1 \wedge \bigwedge_{\mathtt{d}'} (\mathtt{d}' \Rightarrow \bigvee_{\mathtt{d}_2} \mathtt{d}_2)) \quad \text{with} \begin{cases} \mathtt{d}, \mathtt{d}' \in \mathtt{remove}(\rho),\ \mathtt{d}_1, \mathtt{d}_2 \in \mathtt{add}(\rho) \\ \mathtt{d}_1 <_L \mathtt{d} <_L \mathtt{d}_2 <_L \mathtt{d}' \end{cases}$$

In comprises two parts: the first part ($\mathtt{d} \Rightarrow \bigvee_{\mathtt{d}_1}$) ensures that the element ρ is added to the partial variant (by some $\mathtt{d}_1$) before it is removed (by $\mathtt{d}$); the second part ensures that if two delta modules $\mathtt{d}$ and $\mathtt{d}'$ remove ρ, then there is another delta module $\mathtt{d}_2$ in between that adds it.

The constraint for modification of an element ρ simply ensures that ρ is present for the modification:

$$\mathtt{appMOD}(\rho) \triangleq \bigwedge_{\mathtt{d}} \mathtt{d} \Rightarrow (\bigvee_{\mathtt{d}'} \mathtt{d}' \wedge \bigwedge_{\mathtt{d}''} \neg \mathtt{d}'') \quad \text{with} \begin{cases} \mathtt{d} \in \mathtt{modify}(\rho),\ \mathtt{d}'' \in \mathtt{remove}(\rho) \\ \mathtt{d}' \in \mathtt{add}(\rho),\ \mathtt{d}' <_L \mathtt{d}'' <_L \mathtt{d} \end{cases}$$

Basically, it checks that there is a delta module $\mathtt{d}'$ that adds the element before it is modified by $\mathtt{d}$, and that there is no delta module $\mathtt{d}''$ in between that removes it.

The formula $\mathtt{app}(L) \triangleq \bigwedge_{\rho \in \mathtt{add}(L)} \mathtt{appADD}(\rho) \wedge \mathtt{appRM}(\rho) \wedge \mathtt{appMOD}(\rho)$ combines the constraints described above, and the formula $\mathtt{ac}(L) \triangleq L.\mathtt{FMandCK} \Rightarrow \mathtt{app}(L)$ associates to each product of L its applicability constraints. Applicability-consistency (i.e., the fact that variants of L can be generated) is therefore formalized as follows.

Definition 6 (Applicability-consistency). *A product line L is* applicability-consistent *iff the formula* $\mathtt{ac}(L)$ *is valid.*

Product Line Dependency Dependency ensures that no generated variant has a missing dependency, which can be straightforwardly expressed by means of constraints on attributes and classes. For instance, the dependencies induced by "class $\mathtt{C}$ extends class $\mathtt{C}'$" could be encoded with the constraint $\mathtt{decl}(\mathtt{C}) \Rightarrow (\mathtt{decl}(\mathtt{C}') \wedge \neg\mathtt{sub}(\mathtt{C}', \mathtt{C}))$, as the declaration of $\mathtt{C}$ requires that the declaration of $\mathtt{C}'$ is present and that $\mathtt{C}'$ is not a subtype of $\mathtt{C}$ (to ensure that the inheritance graph has no loops). In DOP, since each declaration is made in a module that can be activated or not, dependency constraints must be lifted at the module level. For instance, if the fact that $\mathtt{C}$ extends $\mathtt{C}'$ is declared in the module $\mathtt{d}$, then the previous constraint becomes: $\mathtt{d} \Rightarrow \neg\mathtt{rm}(\mathtt{d}, \mathtt{C}) \Rightarrow \neg\mathtt{modifyEC}(\mathtt{d}, \mathtt{C}) \Rightarrow (\mathtt{decl}(\mathtt{C}') \wedge \neg\mathtt{sub}(\mathtt{C}', \mathtt{C}))$, basically stating that if the module $\mathtt{d}$ is activated and no other module that removes $\mathtt{C}$ or changes its **extends** clause is activated afterward, then the class $\mathtt{C}'$ must be present in the generated variant and must not be a subtype of $\mathtt{C}$.

The product line dependency constraint is generated by exploiting the rules in Figures 3 and 4, which infer a dependency constraint for each expression and declaration, respectively. It is based on the following atomic constraints: $\mathtt{rm}(\mathtt{d}, \mathtt{C})$ (resp. $\mathtt{rm}(\mathtt{d}, \mathtt{C.a})$) ensures that the class $\mathtt{C}$ (resp. attribute $\mathtt{C.a}$) added by the delta module $\mathtt{d}$ will be removed afterward; $\mathtt{modifyEC}(\mathtt{d}, \mathtt{C})$ ensures that the class $\mathtt{C}$ added or modified by the delta module $\mathtt{d}$ will have its **extends** clause modified by another delta module afterward; $\mathtt{replace}(\mathtt{d}, \mathtt{C.m})$ ensures that the method $\mathtt{C.m}$ added or modified by the delta module $\mathtt{d}$ will be replaced by another delta module afterward; $\mathtt{sub}(T, \mathtt{C}')$ ensures that T (either a class or **null**) is a subtype of $\mathtt{C}'$; $\mathtt{decl}(\mathtt{C})$ (resp. $\mathtt{decl}(\mathtt{C.a})$) ensures that the class $\mathtt{C}$ (resp. the attribute $\mathtt{a}$) is present in the generated variant (resp. is an attribute of the class $\mathtt{C}$, possibly through inheritance).

Dependency generation rules for expressions perform a type analysis to know what is the type of each expression, which is used to compute the appropriate

$$\textsc{E:Var}\quad \frac{\mathit{\Gamma}(\mathtt{x}) = \mathtt{C}}{\mathit{\Gamma} \vdash \mathtt{x} : \mathtt{C} \mid \mathbf{true}} \qquad \textsc{E:Field}\quad \frac{\mathit{\Gamma} \vdash e : \mathtt{C} \mid \mathit{\Phi} \qquad \mathtt{FCST}(\mathtt{C.f}) = \mathtt{C}'}{\mathit{\Gamma} \vdash e.\mathtt{f} : \mathtt{C}' \mid \mathit{\Phi} \wedge \mathtt{decl}(\mathtt{C.f})} \qquad \textsc{E:Null}\quad \mathit{\Gamma} \vdash \textsf{\textbf{null}} : \bot \mid \mathbf{true}$$

$$\textsc{E:Meth}\quad \frac{\mathit{\Gamma} \vdash e : \mathtt{C} \mid \mathit{\Phi} \qquad \mathtt{FCST}(\mathtt{C.m}) = \mathtt{C}'(\mathtt{C}_1, \ldots, \mathtt{C}_n) \qquad \mathit{\Gamma} \vdash e_i : T_i \mid \mathit{\Phi}_i \qquad \mathit{\Phi}'_i = \mathtt{sub}(T_i, \mathtt{C}_i)}{\mathit{\Gamma} \vdash e.\mathtt{m}(e_1, \ldots, e_n) : \mathtt{C}' \mid \bigwedge_i (\mathit{\Phi}_i \wedge \mathit{\Phi}'_i) \wedge \mathit{\Phi} \wedge \mathtt{decl}(\mathtt{C.m})} \qquad \textsc{D:New}\quad \mathit{\Gamma} \vdash \textsf{\textbf{new}}\ \mathtt{C}() : \mathtt{C} \mid \mathtt{decl}(\mathtt{C})$$

$$\textsc{E:Cast}\quad \frac{\mathit{\Gamma} \vdash e : T \mid \mathit{\Phi}}{\mathit{\Gamma} \vdash (\mathtt{C})e : \mathtt{C} \mid \mathit{\Phi} \wedge (\mathtt{sub}(T, \mathtt{C}) \vee \mathtt{sub}(\mathtt{C}, T))} \qquad \textsc{E:Assign}\quad \frac{\mathit{\Gamma} \vdash e.\mathtt{f} : \mathtt{C} \mid \mathit{\Phi}_1 \qquad \mathit{\Gamma} \vdash e' : T \mid \mathit{\Phi}_2}{\mathit{\Gamma} \vdash e.\mathtt{f} = e' : \mathtt{C} \mid \mathit{\Phi}_1 \wedge \mathit{\Phi}_2 \wedge \mathtt{sub}(T, \mathtt{C})}$$

Fig. 3. Dependency Generation for Expressions

$$\textsc{D:field}\quad \mathtt{d}, \mathtt{C} \vdash \mathtt{C}'\ \mathtt{f} : \neg\mathtt{rm}(\mathtt{d}, \mathtt{C.f}) \Rightarrow \mathtt{decl}(\mathtt{C}')$$

$$\textsc{D:Meth}\quad \frac{\mathtt{this} : \mathtt{C}; \mathtt{x_i} : \mathtt{C_i} \vdash e : \mathtt{C}' \mid \mathit{\Phi}}{\begin{array}{c}\mathtt{d}, \mathtt{C} \vdash \mathtt{C}_0\ \mathtt{m}(\mathtt{C}_1\ \mathtt{x}_1, \ldots, \mathtt{C}_n\ \mathtt{x}_n)\ \{\textsf{\textbf{return}}\ e\} \\ : \neg(\mathtt{rm}(\mathtt{d}, \mathtt{C.m}) \vee \mathtt{replace}(\mathtt{d}, \mathtt{C.m})) \Rightarrow (\bigwedge_i \mathtt{decl}(\mathtt{C}_i) \wedge \mathit{\Phi} \wedge \mathtt{sub}(\mathtt{C}', \mathtt{C}_0))\end{array}}$$

$$\textsc{D:Class}\quad \frac{\mathtt{d}, \mathtt{C} \vdash AD_i : \mathit{\Phi}_i}{\begin{array}{c}\mathtt{d} \vdash \textsf{\textbf{class}}\ \mathtt{C}\ \textsf{\textbf{extends}}\ \mathtt{C}'\ \{AD_1 \ldots FD_n\} \\ : \neg\mathtt{rm}(\mathtt{d}, \mathtt{C}) \Rightarrow \bigwedge_i \mathit{\Phi}_i \wedge (\neg\mathtt{modifyEC}(\mathtt{d}, \mathtt{C}) \Rightarrow \mathtt{decl}(\mathtt{C}') \wedge \neg\mathtt{sub}(\mathtt{C}', \mathtt{C}))\end{array}} \qquad \textsc{D:ModMD}\quad \frac{\mathtt{d}, \mathtt{C} \vdash MD : \mathit{\Phi}}{\mathtt{d}, \mathtt{C} \vdash \textsf{\textbf{modifies}}\ MD : \mathit{\Phi}}$$

$$\textsc{D:AddAtt}\quad \frac{\mathtt{d}, \mathtt{C} \vdash AD : \mathit{\Phi}}{\mathtt{d}, \mathtt{C} \vdash \textsf{\textbf{adds}}\ AD : \mathit{\Phi}} \qquad \textsc{D:RmAtt}\quad \mathtt{d}, \mathtt{C} \vdash \textsf{\textbf{removes}}\ \mathtt{a} : \mathbf{true} \qquad \textsc{D:RmClass}\quad \mathtt{d} \vdash \textsf{\textbf{removes}}\ \mathtt{C} : \mathbf{true} \qquad \textsc{D:AddClass}\quad \frac{\mathtt{d} \vdash CD : \mathit{\Phi}}{\mathtt{d} \vdash \textsf{\textbf{adds}}\ CD : \mathit{\Phi}}$$

$$\textsc{D:ModClass1}\quad \frac{\mathtt{d}, \mathtt{C} \vdash AO_i : \mathit{\Phi}_i}{\begin{array}{c}\mathtt{d} \vdash \textsf{\textbf{modifies}}\ \mathtt{C}\ \{AO_1 \ldots AO_n\} \\ : \neg\mathtt{rm}(\mathtt{d}, \mathtt{C}) \Rightarrow \bigwedge_i \mathit{\Phi}_i\end{array}} \qquad \textsc{D:ModClass2}\quad \frac{\mathtt{d}, \mathtt{C} \vdash AO_i : \mathit{\Phi}_i}{\begin{array}{c}\mathtt{d} \vdash \textsf{\textbf{modifies}}\ \mathtt{C}\ \textsf{\textbf{extends}}\ \mathtt{C}'\{AO_1 \ldots AO_n\} \\ : \neg\mathtt{rm}(\mathtt{d}, \mathtt{C}) \Rightarrow \bigwedge_i \mathit{\Phi}_i \wedge (\neg\mathtt{modifyEC}(\mathtt{d}, \mathtt{C}) \Rightarrow \mathtt{decl}(\mathtt{C}') \wedge \neg\mathtt{sub}(\mathtt{C}', \mathtt{C}))\end{array}}$$

$$\textsc{D:Delta}\quad \frac{\mathtt{d} \vdash CO_i : \mathit{\Phi}_i}{\vdash \textsf{\textbf{delta}}\ \mathtt{d}\ \{CO_1 \ldots CO_n\} : \mathtt{d} \Rightarrow \bigwedge_i \mathit{\Phi}_i} \qquad \textsc{D:P}\quad \frac{\mathbf{true} \vdash CD_i : \mathit{\Phi}_i \qquad \vdash \mathit{\Delta}_j : \mathit{\Phi}'_j}{\vdash \mathit{\Phi}\ \mathit{\Delta}_1 \ldots \mathit{\Delta}_n\ CD_1 \ldots CD_m : \bigwedge_i \mathit{\Phi}_i \wedge \bigwedge_j \mathit{\Phi}'_j}$$

Fig. 4. Dependency Generation for Declarations

dependency. They have judgments of the form $\mathit{\Gamma} \vdash e : T \mid \mathit{\Phi}$, where: $\mathit{\Gamma}$ is an environment giving the type of each variable; e is the parsed expression; T is its type; and $\mathit{\Phi}$ is the generated dependency constraint. The rules for expressions are quite direct: accessing a variable (rule (E:Var)) does not raise any dependency, while accessing a field requires for this field to be accessible (rule (E:Field)); method calls (rule (E:Meth)) require that the method is accessible and that the parameters have a type consistent with the method's declaration; object creation requires for the class of the object to be defined (rule (E:New)); and **null** does not raise any dependency (rule (E:Null)), while casting and assignment generate constraints ensuring that the right inheritance relation holds (rules (E:Cast) and (E:Assign)).

Dependency generation rules for declarations have judgments of the form $\mathit{\Omega} \vdash A : \mathit{\Phi}$ where $\mathit{\Omega}$ can either be empty, `d` (meaning that we are parsing the content of the module `d`), or `d, C` (meaning that we are parsing the content of the class `C` inside `d`); A is the parsed declaration (e.g., an attribute, a class operation); and Φ is the generated constraint. Rules (D:Field) for field and (D:Meth) for method declarations are quite direct: if the attribute is not removed afterward, the dependencies it generates must be validated. The rule (D:Class) for class declaration is similar (if the class is not removed, its inner dependencies must be validated), with an additional clause for the **extends** clauses (as previously discussed). Rules (D:ModMD) for modifying methods and (D:AddAtt) and (D:AddClass) for adding attributes and classes simply forward the constraints generated from the inner declaration, while removing an attribute or a class (rules (D:RmAtt) and (D:RmClass)) does not generate any dependency. The rules (D:AddClass1) and (D:AddClass2) for modifying a class are simple variations on the rule for class declaration. Finally, the dependencies of a delta module body are activated only if the delta module is activated (rule (D:Delta)), and the dependencies of a whole program is the conjunction of the dependencies of all its parts (rule (D:P)). The resulting constraint thus has the form $\bigwedge_i \mathtt{d}_i \Rightarrow \Phi_i$, giving for all module $\mathtt{d}_i$ its dependencies Φ_i. Let then $\mathtt{dep}(L)$ be the constraint generated for the product line L. The formula $\mathtt{dc}(L) \triangleq L.\mathtt{FMandCK} \Rightarrow \mathtt{dep}(L)$ associates to each product of L its dependency constraints. Dependency-consistency (i.e., variants of L have all their dependencies fulfilled) is therefore formalized as follows.

Definition 7 (Dependency-consistency). *A product line L is* dependency-consistent *iff the formula $\mathtt{dc}(L)$ is valid.*

Correctness and Completeness of the Approach The following theorem states that, if the product line follows Guideline G2, then the presented IFΔJ product line type checking approach is correct with respect to generating variants and checking them using the IFJ type system. The approach is complete (i.e., if the check performed by the approach fails then at least one variant is not a well-typed IFJ program) if also Guideline G1 is followed.

Theorem 1. *Let L be a type-uniform product line. Consider the properties:*
i. L is well partially-typed, applicability- and dependency-consistent.
ii. Variants of L can be generated and are well-typed IFJ programs.
Then: (i) implies (ii); and if L has no useless operations then (ii) implies (i).

5 Type Checking for IFΔJ without Guidelines

In this section we outline how the type checking approach presented in Section 4 can be tuned to non type-uniform product lines (i.e., not to exploit any guidelines). This modification is quite straightforward, although it involves many technical details. Partial typing must be adapted since the product line FCST maps attribute names to sets of types with possibly more than one element, and expressions can have more than one type. E.g., a method call expression $e.\mathtt{m}(\overline{e'})$ can use any declaration of the method $\mathtt{C.m}$ (considering that e is typed $\mathtt{C}$) whose type accepts a combination of types of the call's arguments. So partial typing may carry a combinatorial explosion.

Applicability does not need any modification to analyze non-uniform programs. This is due to the fact that the applicability criteria focuses on the interplay between delta operations and do not consider attribute types.

Dependency is the part that changes more: it now has to be type-aware, and thus subsumes partial typing. We illustrate it on the rule that generates the dependency for field usage (second rule in Fig. 3). This rule must be extended in two ways to manage non-uniform programs: (i) e can have more than one type; (ii) the field type lookup `FCST(C.f)` can return different possible types for `C.f`, depending on which modules are activated. Consequently, the dependency generation judgment for expressions now has the form $\Gamma \vdash e : [\Phi_i \mapsto T_i]_{i \in I}$ where T_i are the possible types of e, and Φ_i is the condition (i.e. which module must or must not be activated) for e to have the type T_i in the final product. Hence, the rule becomes as displayed on the right,

$$\frac{\Gamma \vdash e : [\Phi_i \mapsto \mathtt{C}_i]_i \cup [\Phi_{i'} \mapsto \bot]_{i'} \qquad \mathtt{FCST}(\mathtt{C}_i.\mathtt{f}) = [\Phi_{i,j} \mapsto \mathtt{C}_{i,j}]_j}{\Gamma \vdash e.\mathtt{f} : [\Phi_i \wedge \Phi_{i,j} \mapsto \mathtt{C}_{i,j}]_{i,j}}$$

where $\Phi_{i,j}$ is the formula that enforces that the field `f` accessible from the class $\mathtt{C}_i$ has the type $\mathtt{C}_{i,j}$ in the final product.

Correctness and completeness are stated as in Theorem 1 by dropping the assumption that the product line is type-uniform.

6 Three other Guidelines

Our type-checking approach is modularized in three parts: i) partial typing performs a preliminary type analysis that can be exploited by an IDE for prompt notification of type-errors and auto-completing code; ii) applicability ensures that variants can be generated; and iii) dependency completes the analysis done by the partial typing. The approach is tunable to exploit DOP guidelines that enforce structural regularities in product line implementation. In Section 4 we have presented a version tuned to exploit type-uniformity. In this section we briefly discuss three other automatically checkable guidelines (other useful guidelines could be devised).

First, whenever it is possible to enforce the following guideline (satisfied by the EPL), the dependency analysis can be simplified, as it is no longer needed to check the absence of inheritance loop in the generated variant (cf. dependency generation for class declaration and modification in Figure 4).

G3 Ensure that the product line FCST subtyping relation is acyclic.

If a product line cannot be made variant-type-uniform, then guideline G2 cannot be enforced (see Proposition 1), and understanding the structure of the SPL may become an issue. The following guideline (satisfied by the EPL) 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.

Consider for instance a module `d` that declares an attribute `C.a` with a type t. Then, if the SPL follows G3, we are sure that each variant using `d` in its construction will have `C.a` typed t when it contains this attribute.

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 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 the more efficient type checking technique of Section 4 can be used (even if the whole product line is not type-uniform). 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.

The goal of this guideline is to allow to use as much as possible the version of the approach presented in Section 4. For the EPL the partition that satisfies Guideline G5.i is unique: the two products with feature fEval1 and the four products with feature fEval2. However, in general, such a partition may be not unique and tool support for identifying a partition that satisfies G5.i and further conditions (e.g., having a minimal number of parts) or G5.ii and other conditions (e.g., the number of products in the non type-uniform part is as small as possible) would be valuable.

7 Related Work

Product line analysis approaches can be classified into three main categories [23]: *Product-based* analyses operate only on generated variants (or models of variants); *Family-based* analyses operate only on the AB by exploiting the FM and the CK to obtain results about all variants; *Feature-based* analyses operate on the building blocks of the different variants (feature modules in FOP and delta modules in DOP) in isolation (without using the FM and the CK) to derive results on all variants. We refer to [23] for a survey on product line type checking. Here we discuss previous type checking approaches for DOP [5, 8] and the two approaches for FOP that are closets to our proposal [9, 22].

The type checking approach for DOP in [5] comprises: a feature-based analysis that uses a constraint-based type system for IFJ to infer a type abstraction for each delta module; and a product-based step that uses these type abstractions to generate, for each product of the SPL, a type abstraction (of the associated variant) that is checked to establish whether the associated variant type checks. The approach of [5] is enhanced in [8] by introducing a family-based step that builds a product family generation tree which is then traversed in order to perform optimized generation and check of type abstractions of all variants. The approach proposed in this paper, which is feature-family-based, represents an achievement over [5, 8] since it does not require to iterate over the set of products (cf. Section 1) and supports earlier detection of errors via partial typing.

The paper [22] informally illustrates the implementation of a family-based approach for the AHEAD system [4]. The approach comprises: i) a family-feature-based step that computes for each class a stub (all stubs can be understood as a type-uniform FCST for the product line) and compiles each feature module in the context of all stubs (thus performing checks corresponding to our type-uniformity and partial-typing); and ii) a family-based step that infers a set of constraints that are combined with the FM to generate a formula (corresponding to our type-uniform applicability and dependency) whose satisfiability should imply that all variants successfully compile.

The paper [9] formalizes a feature-family-based approach for the Lightweight Feature Java (LFJ) calculus, which models FOP for the Lightweight Java (LJ) [21] calculus. The approach comprises: i) a feature-based step that uses a constraint-based type system for LFJ to analyze each feature module in isolation and infer a set of constraints for each feature module; and ii) a family-based step where the FM and the previously inferred constraints are used to generate a formula whose satisfiability implies that all variants type check. The applicability and dependency analyses presented in Section 5 provide an extension to DOP of these two steps. Moreover, our approach provides partial typing for early error detection and is tunable to exploit different programming guidelines.

8 Conclusions and Future Work

We have proposed a modular and tunable approach for type checking DOP product lines. A prototypical implementation is available [2] (currently only the version of the approach exploiting type-uniformity is supported).

In future work we plan to: implement our approach for both DeltaJ 1.5 [14] (a prototypical implementation of DOP that supports full Java 1.5) and ABS [12] (this would allow experimental comparison with the approaches of [5, 8], which have been implemented for ABS [1]); to develop case studies to evaluate the effectiveness of the approach and of the proposed guidelines; to investigate further DOP guidelines; and to develop tool support to allow the programmer to choose the guidelines to be automatically enforced. We also plan to investigate whether the proposed DOP guidelines (or other guidelines) could be useful for other kind of product line analyses. In particular, we would like to consider formal verification (proof systems for the verification of DOP product lines of Java programs have been recently proposed [10, 7]).

Acknowledgements We are grateful to Don Batory for clarifications about previous work on type checking FOP, and to Ina Schaefer and Thomas Thüm for discussions about how to classify SPL type checking approaches. We also thank the iFM 2016 anonymous reviewers for insightful comments and suggestions.

References

1. https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta.
2. https://github.com/gzoumix/IFDJTS.
3. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. of SPLC 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. of ICSE 2003*, pages 187–197. IEEE, 2003.
5. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.
7. F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for delta-oriented programming. In *Proc. of SPLC 2012 - Volume 2*, pages 53–60. ACM, 2012.
8. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *Proc. of ISoLA*, volume 7609 of *LNCS*. Springer, 2012.
9. B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. of ESEC/FSE 2009*. ACM, 2009.
10. R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In *Proc. of ISoLA*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012.
11. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
12. 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.
13. C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In *Proc. of SPLC 2009*, pages 181–190, 2009.
14. J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. Deltaj 1.5: Delta-oriented programming for java 1.5. In *Proc. of PPPJ 2014*, pages 63–74. ACM, 2014.
15. C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.
16. M. Lienhardt and D. Clarke. Conflict detection in delta-oriented programming. In *ISoLA*, pages 178–192, 2012.
17. 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.
18. I. Schaefer. Proc. of VaMoS '10. In *Intl. Workshop on Variability Modelling of Software-intensive Systems*, 2010.
19. 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.
20. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Proc. of FOSD 2010*, pages 49–56. ACM, 2010.
21. R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proc. of OOPSLA 2007*, pages 499–514. ACM, 2007.
22. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. of GPCE '07*, pages 95–104. ACM, 2007.
23. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 2014.