On Slicing Software Product Line Signatures

Ferruccio Damiani, Michael Lienhardt, Luca Paolini

On Slicing Software Product Line Signatures

Ferruccio Damiani[1], Michael Lienhardt[2], and Luca Paolini[1]

[1] University of Turin, Italy
{ferruccio.damiani, luca.paolini}@unito.it
[2] ONERA, Palaiseau, France
michael.lienhardt@onera.fr

Abstract. A Software Product Line (SPL) is a family of similar programs (called variants) generated from a common artifact base. Variability in an SPL can be documented in terms of abstract description of functionalities (called features): a feature model (FM) identifies each variant by a set of features (called a product). Delta-orientation is a flexible approach to implement SPLs. An SPL Signature (SPLS) is a variability-aware Application Programming Interface (API), i.e., an SPL where each variant is the API of a program. In this paper we introduce and formalize, by abstracting from SPL implementation approaches, the notion of slice of an SPLS K for a set of features F (i.e., an SPLS obtained from by K by hiding the features that are not in F). Moreover, we formulate the challenge of defining an efficient algorithm that, given a delta-oriented SPLS K and a set of features F, returns a delta-oriented SPLS that is an slice of K for F. Thus paving the way for further research on devising such an algorithm. The proposed notions are formalized for SPLs of programs written in an imperative version of Featherweight Java.

1 Introduction

A *Software Product Line* (SPLs) is a family of similar programs, called *variants*, that have a well-documented variability and are generated from a common artifact base [10, 39, 3]. An SPL can be structured into: (i) a *feature model* describing the variants in terms of *features* (each feature is a name representing an abstract description of functionality and each variant is identified by a set of features, called a *product*); (ii) an *artifact base* comprising language dependent reusable code artifacts that are used to build the variants; and (iii) *configuration knowledge* connecting feature model and artifact base by specifying how, given a product, the corresponding a variant can be derived from the code artifacts—thus inducing a mapping from products to variants, called the *generator* of the SPL.

An interface can be understood as a partial specification of the functionalities of a system. Such a notion of interface provides a valuable support for modularity. If a system can be decomposed in subsystems in such a way that all the uses of each subsystem by the other subsystems are mediated by interfaces of the subsystem, then subsystem changes that do not break the interfaces are transparent (with respect to the specifications expressed by the interfaces) to the other subsystems.

In this paper we formalize, by abstracting from SPL implementation approaches, the problem of designing an efficient algorithm that, given an SPL and subset F of it features, extracts an interface for the SPL that exposes only the functionalities associated to the features in F. We build on the notions of signature and interface of an SPL introduced in [17] (see also [19]). An *SPL Signature* (SPLS) is a variability-aware *Application Programming Interface* (API), i.e., an SPL where each variant is a program API. The *signature of an SPL* $\mathtt{L}$ is an SPLS $\mathtt{Z}$ where: (i) the features are the same of $\mathtt{L}$; (ii) the products are the same of $\mathtt{L}$; and (iii) each variant is the *program signature* (i.e., a program API that exposes all the functionalities) of the corresponding variant of $\mathtt{L}$. An SPLS $\mathtt{Z}_1$ is:

- an *interface of an SPLS* $\mathtt{Z}_2$ iff[3] (i) the features of $\mathtt{Z}_1$ are are a subset of the features of $\mathtt{Z}_2$; (ii) the products of $\mathtt{Z}_1$ are obtained for the products of $\mathtt{Z}_2$ by dropping the features that are not in $\mathtt{Z}_1$; and (iii) for each product p_1 of $\mathtt{Z}_1$, its associated variant is an interface of all the variants associated to the products of $\mathtt{Z}_2$ from which p_1 can be obtained by dropping the features that are not in $\mathtt{Z}_1$; and
- an *interface of an SPL* $\mathtt{L}$ iff it is an interface of the signature of $\mathtt{L}$.

The contribution of this paper is twofold.

1. We introduce and formalize, by abstracting from SPL implementation approaches, the notion of *slice of an SPLS for a set of features $\mathcal{F}$. It* lifts to SPLs the notion of *slice of a FM* introduced in [1] (see also [43]). Namely, we define an operator that given an SPLS $\mathtt{Z}$ and a set of features $\mathcal{F}$ returns an SPLS that has exactly the features in $\mathcal{F}$ and is an interface of $\mathtt{Z}$.
2. We introduce and formalize the problem of devising a feasible algorithm that takes as input a delta-oriented SPLS $\mathtt{Z}$ [19] and a set of features $\mathcal{F}$, and yields as output a delta-oriented SPLS that is a slice of $\mathtt{Z}$ for $\mathcal{F}$. Thus paving the way for further research on devising such an algorithm.

Since extracting the signature of a delta-oriented SPL is quite straightforward [17], an algorithm like the one described in point 2 above would provide a way to extract from an SPL an interface that exposes only the functionalities associated to a given set of features. This would enable refactoring a delta-oriented SPL by decomposing it into a *Multi SPL* (MPL), that is a set of interdependent SPLs that need to be managed in a decentralized fashion by multiple teams and stakeholders [29], and performing compositional analysis of delta-oriented MPLs [19]. Moreover, the implementation-independent formalization described in point 1 above might foster further research on MPLs comprising SPLs implemented according to different approaches (see, e.g., [42, 47, 3] for a presentation of different SPL implementation approaches).

[3] In [19] the phrase "subsignature of an SPLS" is used instead of "interface of an SPLS".

Organisation of the Paper. Section 2 provides the necessary background on SPLs, SPLSs and interfaces. Section 3 provides a definition of the SPLS slice operator that abstracts from SPL implementation approaches. Section 4 recalls delta-oriented SPLs and illustrates the problem of devising a feasible algorithm for slicing delta-oriented SPLSs. Related work is discussed in Section 5, and Section 6 concludes the paper by outlining possible future work.

2 A Recollection of SPLs, SPL Signatures and Interfaces

2.1 Feature Models, Feature Module Slices and Interfaces

The following definition provides an extensional account on the notion of feature model, namely a feature model is represented as a pair "(set of features, set of products)", thus allowing to abstract from implementation approaches—see e.g. [4] for a discussion on possible representations of feature models.

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

The slice operator for feature models introduced by Acher et al. [1], given a feature model $\mathcal{M}$ and a set of features Y, returns the feature model obtained from $\mathcal{M}$ by removing the features not in Y.

Definition 2 (Feature model slice operator). *Let* $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ *be a feature model. The* slice *operator* Π_Y *on feature models, where* Y *is a set of features, is defined by:* $\Pi_Y(\mathcal{M}) = (\mathcal{F} \cap Y, \{p \cap Y \mid p \in \mathcal{P}\})$.

More recently, Schröter et al. [43] introduced the slice function $\mathbf{S}$ such that $\mathbf{S}(\mathcal{M}, Y) = \Pi_{\mathcal{F}\setminus Y}(\mathcal{M})$. Schröter et al. [43] also introduced the following notion of feature model interface.

Definition 3 (Interface relation for feature models). *A feature model* $\mathcal{M}_0 = (\mathcal{F}_0, \mathcal{P}_0)$ *is an* interface *of feature model* $\mathcal{M} = (\mathcal{F}, \mathcal{P})$*, denoted as* $\mathcal{M}_0 \preceq \mathcal{M}$*, whenever both* $\mathcal{F}_0 \subseteq \mathcal{F}$ *and* $\mathcal{P}_0 = \{p \cap \mathcal{F}_0 \mid p \in \mathcal{P}\}$ *hold.*

It is worth observing that $\mathcal{M}_0 \preceq \mathcal{M}$ holds if and only if $\mathcal{M}_0 = \Pi_Y(\mathcal{M})$, where Y are features of $\mathcal{M}_0$. Moreover, the interface relation for feature models is reflexive, transitive and anti-symmetric.

Given a product p, we define $\mathcal{I}_p(x) = \begin{cases} \mathbf{true} & \text{if } x \in p, \\ \mathbf{false} & \text{otherwise} \end{cases}$ and, given a *propositional formula over features* (i.e., where propositional variable are feature names) ϕ, we wrote $\mathcal{I}_p \models \phi$ to mean that ϕ evaluates to **true** by replacing its variables according to $\mathcal{I}_p$.

Example 1 (Slicing the Expression Feature Model). Consider the propositional formula $\phi_{\text{EPL}} = \mathsf{Lit} \wedge (\mathsf{Print} \vee \mathsf{Eval})$. The feature model $\mathcal{M}_{\text{EPL}} = (\mathcal{F}_{\text{EPL}}, \mathcal{P}_{\text{EPL}})$ has 4 features $\mathcal{F}_{\text{EPL}} = \{\mathsf{Lit}, \mathsf{Add}, \mathsf{Print}, \mathsf{Eval}\}$ and 8 products $\mathcal{P}_{\text{EPL}} = \{\ p \mid p \subseteq$

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

Fig. 1: IFJ programs

$\mathcal{F}_{\mathrm{EPL}}$ and $\mathcal{I}_p \models \phi_{\mathrm{EPL}}\}$. It describes a family of programs implementing an expression datatype where Literals are mandatory and Additions are optional, and where either a Print operation or an Evaluation operation must be supported. Let $\mathcal{F}_{\mathrm{EPL}_0} = \{\mathsf{Lit}, \mathsf{Add}, \mathsf{Print}\}$, then we have that $\Pi_{\mathcal{F}_{\mathrm{EPL}_0}}(\mathcal{M}_{\mathrm{EPL}}) = \mathcal{M}_{\mathrm{EPL}_0} = (\mathcal{F}_{\mathrm{EPL}_0}, \mathcal{P}_{\mathrm{EPL}_0})$, where $\mathcal{P}_{\mathrm{EPL}_0} = \{\, p_{\mathrm{I}}, p_{\mathrm{II}}, p_{\mathrm{III}}, p_{\mathrm{IV}} \,\}$ with $p_{\mathrm{I}} = \{\mathsf{Lit}\}$, $p_{\mathrm{II}} = \{\mathsf{Lit}, \mathsf{Add}\}$, $p_{\mathrm{III}} = \{\mathsf{Lit}, \mathsf{Print}\}$, $p_{\mathrm{IV}} = \mathcal{F}_{\mathrm{EPL}_0}$.

2.2 SPLs of IFJ programs

Imperative Featherweight Java (IFJ) [7] is an imperative version of *Featherweight Java* (FJ) [30]. The abstract syntax of IFJ *programs* is given in Figure 1. Following Igarashi et al. [30], we use the overline notation for (possibly empty) sequences of elements—e.g., $\overline{e}$ stands for a sequence of expressions $e_1, \ldots, e_n$ ($n \geq 0$)—and we denote the empty sequence by $\emptyset$.

A program P is a sequence of class declarations $\overline{CD}$. A class declaration comprises the name `C` of the class, the name of the superclass (which must always be specified, even if it is the built-in class `Object`) and a list of attribute (field or method) declarations $\overline{AD}$. Variables `x` include the special variable `this` (implicitly bound in any method declaration *MD*), which may not be used as the name of a method's formal parameter. All fields and methods are public, there is no field shadowing, there is no method overloading, and each class is assumed to have an implicit constructor that initialized all fields to **null**.

An *attribute name* `a` is either a field name `f` or a method name `m`. Given a program P, a class name `C` and an attribute name `a`, we write $\mathrm{dom}(P)$, $P(\mathtt{C})$, $\mathrm{dom}_P(\mathtt{C})$, $\leq:_P$, $CD(\mathtt{a})$, and $lookup_P(\mathtt{a}, \mathtt{C})$ to denote, respectively: the set of class names declared in P; the declaration of `C` in P when it exists; the set of attribute names declared in $P(\mathtt{C})$; the subtyping relation in P (i.e., the reflexive and transitive closure of the immediate **extends** relation); the declaration of attribute `a` in *CD*; and the declaration of the attribute `a` in the closest superclass of `C` (including `C` itself) that contains a declaration for `a` in P, when it exists. We write $<:_P$ to denote the strict subtyping relation in P, defined by: $\mathtt{C}_1 <:_P \mathtt{C}_2$ if and only if $\mathtt{C}_1 \leq:_P \mathtt{C}_2$ and $\mathtt{C}_1 \neq \mathtt{C}_2$.

As usual, we identify two IFJ programs P_1 and P_2 (written $P_1 = P_2$) up to: (i) the order of class declarations and attribute declarations, and (ii) renaming of the formal parameters of methods. The following notational convention entails the assumption that the classes declared in a program have distinct names, the attributes declared in a class have distinct names, and the formal parameter declared in a method have distinct names.

Convention 1 (On sequences of named declarations) *Whenever we write a sequence of named declarations $\overline{N}$ (e.g., classes, attributes, parameters, etc.) we assume that they have pairwise distinct names. We write names$(\overline{N})$ to denote the sequence of the names of the declarations in $\overline{N}$. Moreover, when no confusion may arise, we sometimes identify sequences of pairwise distinct elements with sets, e.g., we write $\overline{e}$ as short for $\{e_1, \ldots, e_n\}$.*

We require that every IFJ program P satisfies the following *sanity conditions*:

SC1: For every class name `C` (except `Object`) appearing anywhere in P, we have $\mathtt{C} \in \mathrm{dom}(P)$.
SC2: The strict subtyping relation $<:_P$ is acyclic.
SC3: If $\mathtt{C}_2 <:_P \mathtt{C}_1$, then $\mathrm{dom}(P(\mathtt{C}_1)) \cap \mathrm{dom}(P(\mathtt{C}_2))$ does not contain field names.
SC4: If $\mathtt{C}_2 <:_P \mathtt{C}_1$ then for all method names $\mathtt{m} \in \mathrm{dom}(P(\mathtt{C}_1)) \cap \mathrm{dom}(P(\mathtt{C}_2))$ the methods $P(\mathtt{C}_1)(\mathtt{m})$ and $P(\mathtt{C}_2)(\mathtt{m})$ have the same header (up to renaming of the formal parameters).

Note that **SC3** and **SC4** formalize the requirements "there is no field shadowing" and "there is no method overloading", respectively. Type system, operational semantics, and type soundness for IFJ are given in [7].

Remark 1 (Sugared IFJ syntax). To improve readability, in the examples we use Java syntax for field initialization, primitive data types, strings and sequential composition. Encoding in IFJ syntax a program written in such a *sugared IFJ syntax* is straightforward (see [7]).

Example 2 (The Expression Program). Figure 2 illustrates a sugared IFJ program called the Expression Program (EP for short), that encodes the following grammar of numerical expressions:

$$\mathtt{Exp} ::= \mathtt{Lit} \mid \mathtt{Add} \qquad \mathtt{Lit} ::= \text{non-negative-integers} \qquad \mathtt{Add} ::= \mathtt{Exp}\ \text{“+”}\ \mathtt{Exp}$$

The EP consists of: (i) a class `Exp` representing all expressions; (ii) a class `Lit` representing literals; and, (iii) a class `Add` representing an addition between two expressions. All these classes implement a method `toInt` that computes the value of the expression, and a method `toString` that gives a textual representation of the expression. Note that the concept of expression is too general to provide a meaningful implementation of these methods, and thus the class `Exp` is supposed to be used as a type and should never be instantiated.

The following definition (taken form [35]) provides an extensional account on the notion of SPL, thus allowing to abstract from implementation approaches—see e.g. [42, 47] for a survey on SPL implementation approaches.

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

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

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

Fig. 2: The Expression Program

Definition 4 (SPL, extensional representation). *An SPL* $\mathtt{L}$ *is a pair* $(\mathcal{M}_{\mathtt{L}}, \mathcal{G}_{\mathtt{L}})$ *where* $\mathcal{M}_{\mathtt{L}} = (\mathcal{F}_{\mathtt{L}}, \mathcal{P}_{\mathtt{L}})$ *is the feature model of the SPL and* $\mathcal{G}_{\mathtt{L}}$ *is the generator of the SPL, i.e., a function from the products in* $\mathcal{P}_{\mathtt{L}}$ *to the variants.*[4]

Type system, operational semantics, and type soundness for IFJ are given in [7]. We say that the extensional representation of an SPL of IFJ programs is well typed to mean that the variants are well-typed IFJ programs.

Example 3 (The Expression Product Line). The Expression Product Line (EPL) is the SPL EPL $= (\mathcal{M}_{\mathrm{EPL}}, \mathcal{G}_{\mathrm{EPL}})$ where $\mathcal{M}_{\mathrm{EPL}} = (\mathcal{F}_{\mathrm{EPL}}, \mathcal{P}_{\mathrm{EPL}})$ is as in Example 1, $\mathcal{G}_{\mathrm{EPL}}(\mathcal{F}_{\mathrm{EPL}})$ is the program EP in Example 2, and the other 7 variants are obtained from EP by dropping class `Add` whenever feature Add is not selected and by dropping methods `toString` and `toInt` whenever features Print and Eval are not selected, respectively.

2.3 Signatures and Interfaces for SPLs of IFJ Programs

The abstract syntax of IFJ *program signatures* is given Figure 3. From a syntactic perspective, a program signature is essentially a program deprived of method bodies, and a class signature is a class deprived of method bodies. The *signature of a program P*, denoted as ***signature***(P), is the program signature obtained from P by dropping the body of its methods.

Remark 2 (On the signature of a sugared IFJ program). The signature of a program written in sugared IFJ syntax (introduced Remark 1) is obtained by dropping the body of the methods and the initialization of the field declarations. Notably, the signature of a sugared IFJ program is an IFJ program signature.

Given a program signature PS, a class name $\mathtt{C}$, a class signature CS and an attribute name $\mathtt{a}$, we write $\mathrm{dom}(PS)$, $PS(\mathtt{C})$, $\mathrm{dom}_{PS}(\mathtt{C})$, $\leq:_{PS}$, $CS(\mathtt{a})$, and $lookup_{PS}(\mathtt{a}, \mathtt{C})$ to denote, respectively: the set of class names declared in PS; the declaration of the class signature of $\mathtt{C}$ in PS when it exists; the set of attribute

[4] In [35] the generator is modeled as a partial function in order to encompass ill-formed SPLs where, for some product, the generation of the associated variant fails. In this paper we focus on well-formed SPLs, so we consider a total generator.

$\mathit{PS} ::= \overline{\mathit{CS}}$	Program Signature
$\mathit{CS} ::= \textsf{class}\ \mathtt{C}\ \textsf{extends}\ \mathtt{C}\ \{\ \overline{\mathit{AS}}\ \}$	Class Signature
$\mathit{AS} ::= \mathit{FD} \mid \mathit{MH}$	Attribute (Field or Method) Signature

Fig. 3: IFJ program signatures

names declared in $\mathit{PS}(\mathtt{C})$; the subtyping relation in PS; the set of attribute names declared in CS; and the signature of the attribute $\mathtt{a}$ in the closest supertype of $\mathtt{C}$ (including itself) that contains a declaration for $\mathtt{a}$ in PS, when it exists. We write $<:_{\mathit{PS}}$ to denote the strict subtyping relation in PS, defined by: $\mathtt{C}_1 <:_{\mathit{PS}} \mathtt{C}_2$ if and only if $\mathtt{C}_1 \leq:_{\mathit{PS}} \mathtt{C}_2$ and $\mathtt{C}_1 \neq \mathtt{C}_2$.

We require that every IFJ program signature PS satisfies the *sanity conditions* listed below.

SCi: For every class name $\mathtt{C}$ (except $\mathtt{Object}$) appearing in an **extends** clause in PS, we have $\mathtt{C} \in \mathrm{dom}(\mathit{PS})$.
SCii: The strict subtyping relation $<:_{\mathit{PS}}$ is acyclic.
SCiii: If $\mathtt{C}_2 <:_{\mathit{PS}} \mathtt{C}_1$, then for all attributes $\mathtt{a} \in \mathrm{dom}(\mathit{PS}(\mathtt{C}_1)) \cup \mathrm{dom}(\mathit{PS}(\mathtt{C}_2))$ we have $\mathit{PS}(\mathtt{C}_1)(\mathtt{a}) = \mathit{PS}(\mathtt{C}_2)(\mathtt{a})$.

It is worth noticing that sanity condition **SCi** is weaker than **SC1**: a program signature is not required to provide a declaration for the class names occurring in attribute declarations. Recall that in IFJ field shadowing if forbidden (cf. sanity condition **SC3**). For the sake of simplicity, in program signatures there is no such a restriction: field and method signatures are treated uniformly.

A program signature PS can be understood as an API that expresses requirements on programs. I.e., *program signature PS is an interface of program P* if P provides at least all the classes, attributes and subtyping relations in PS. Similarly, *program signature PS is an interface*[5] *of program signature PS_0* if PS_0 provides at least all the classes, attributes and subtyping relations in PS. These notions are formalized by the following definitions.

Definition 5 (Interface relation for program signatures). *A program signature PS_1 is an* interface *of a program signature PS_2, denoted as $\mathit{PS}_1 \preceq \mathit{PS}_2$, iff: (i) $\mathrm{dom}(\mathit{PS}_1) \subseteq \mathrm{dom}(\mathit{PS}_2)$; (ii) $\leq:_{\mathit{PS}_1} \subseteq \leq:_{\mathit{PS}_2}$; and (iii) for all class name $\mathtt{C} \in \mathrm{dom}(\mathit{PS}_1)$, for all attribute $\mathtt{a}$, we have that if $\mathit{lookup}_{\mathit{PS}_1}(\mathtt{a},\mathtt{C})$ is defined then $\mathit{lookup}_{\mathit{PS}_2}(\mathtt{a},\mathtt{C})$ is defined and $\mathit{lookup}_{\mathit{PS}_1}(\mathtt{a},\mathtt{C}) = \mathit{lookup}_{\mathit{PS}_2}(\mathtt{a},\mathtt{C})$.*

Definition 6 (Interface relation between signatures and programs). *A program signature PS is an* interface *of program P, denoted as $\mathit{PS} \preceq P$, iff $\mathit{PS} \preceq \textbf{\textit{signature}}(P)$ holds.*

The interface relation for program signatures is a preorder. Namely, it is reflexive (which implies $\textbf{\textit{signature}}(P) \preceq P$), transitive, and (due to the possibility of overriding of attribute signatures) not antisymmetric (i.e., $\mathit{PS}_1 \preceq \mathit{PS}_2$

[5] In [19] the word “subsignature” is used instead of “interface”.

and $PS_2 \preceq PS_1$ do not imply $PS_1 = PS_2$). Since $\preceq$ is a preorder, the relation $\approxeq = (\preceq \cap \succeq)$ is an equivalence relation, and the relation $\preceq$ can be understood as a partial order (reflexive, transitive and antisymmetric) on the set of $\approxeq$-equivalence classes. The (equivalence class of the) empty program signature $\emptyset$ is the bottom element with respect to $\preceq$.

Example 4 (Signature and interfaces of the Expression Program). Let EP be the program illustrated in Figure 2. Given the following three signatures

EPS=

```
class Exp extends Object {
  String name;
  Int toInt(); String toString();
}
class Lit extends Exp {
  Int val; Lit setLit(Int x);
  Int toInt(); String toString();
}
class Add extends Exp {
  Exp a; Exp b;
  Int toInt(); String toString();
}
```

ESP$_1$ =

```
class Exp extends Object {
  String name;
  Int toInt(); String toString();
}
class Lit extends Exp {
  Int val; Lit setLit(Int x);
}

class Add extends Exp {
  Exp a; Exp b;
}
```

EPS$_2$ =

```
class Exp extends Object {
  String name;
  String toString();
}
class Lit extends Exp {
  String name;
  Int toInt();
}
class Add extends Object {
  String name;
  Exp a;
}
```

we have: EPS = $\boldsymbol{signature}$(EP), EPS$_1 \approxeq$ EPS, EPS$_2 \preceq$ EPS, and EPS $\not\preceq$ EPS$_2$.

The notion of *SPL signature* (SPLS) [19] describes the API of an SPL, i.e., the APIs of the variants generated by the SPL. Namely, an SPLS is an SPL where the variants are program signatures instead of programs. The following definition provides an extensional account of this notion.

Definition 7 (SPLS, extensional representation). *An SPLS* $\mathtt{Z}$ *is a pair* $(\mathcal{M}_{\mathtt{Z}}, \mathcal{G}_{\mathtt{Z}})$ *where* $\mathcal{M}_{\mathtt{Z}} = (\mathcal{F}_{\mathtt{Z}}, \mathcal{P}_{\mathtt{Z}})$ *is the feature model of the SPLS and* $\mathcal{G}_{\mathtt{Z}}$ *is the generator of the SPLS, i.e., a mapping from the products in* $\mathcal{P}_{\mathtt{Z}}$ *to variant signatures.*

The notion of *signature of an SPL* [19] naturally lifts that of signature of a program. Namely, the *signature of an SPL* $\mathtt{L} = (\mathcal{M}_{\mathtt{L}}, \mathcal{G}_{\mathtt{L}})$ is the SPLS defined by $\boldsymbol{signature}(\mathtt{L}) = (\mathcal{M}_{\mathtt{L}}, \boldsymbol{signature}(\mathcal{G}_{\mathtt{L}}))$, where $\boldsymbol{signature}(\mathcal{G}_{\mathtt{L}})$ is defined by

$$\boldsymbol{signature}(\mathcal{G}_{\mathtt{L}})(p) = \boldsymbol{signature}(\mathcal{G}_{\mathtt{L}}(p)), \text{ for all } p \in \mathcal{P}_{\mathtt{L}}.$$

The notion of *interface of an SPLS* [19] naturally lifts the one of interface of a program signature (in Definition 5) by combining it with the notion of feature model interface (in Definition 3).

Definition 8 (Interface relation for SPLSs). *An SPLS* $\mathtt{Z}_1$ *is a* interface *of an SPLS* $\mathtt{Z}_2$*, denoted as* $\mathtt{Z}_1 \preceq \mathtt{Z}_2$*, iff: (i)* $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$*; and (ii) for each* $p \in \mathcal{P}_{\mathtt{Z}_2}$*,* $\mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1}) \preceq \mathcal{G}_{\mathtt{Z}_2}(p)$*.*

Similarly, the notion of *interface of an SPL* lifts the notion interface of a program (in Definition 6).

Definition 9 (Interface relation between SPLs and SPLSs). *An SPLS* $\mathtt{Z}$ *is an* interface *of an SPL* $\mathtt{L}$*, denoted as* $\mathtt{Z} \preceq \mathtt{L}$*, iff* $\mathtt{Z} \preceq \boldsymbol{signature}(\mathtt{L})$ *holds.*

It is worth observing that the interface relation for SPLSs has two degrees of freedom: it allows to hide features from the feature model (as described in Definition 3), and it allows to hide declarations from the SPLS variants (as described in Definition 5). Additionally, note that the interface relation for SPLSs, like the one for program signatures (see the explanation after Definition 6), is reflexive, transitive and not anti-symmetric. We say that two SPLSs $\mathtt{Z}_1$ and $\mathtt{Z}_2$ are *equivalent*, denoted as $\mathtt{Z}_1 \approxeq \mathtt{Z}_2$, to mean that both $\mathtt{Z}_1 \preceq \mathtt{Z}_2$ and $\mathtt{Z}_2 \preceq \mathtt{Z}_1$ hold.

Example 5 (Signature and interfaces of the EPL). Consider the signature of the EPL of Example 3: $\boldsymbol{signature}(\mathrm{EPL}) = \mathrm{EPLS} = (\mathcal{M}_{\mathrm{EPLS}}, \mathcal{G}_{\mathrm{EPLS}})$. Let $\mathrm{EPLS}' = (\mathcal{M}_{\mathrm{EPL}}, \mathcal{G}_{\mathrm{EPLS}'})$, where $\mathcal{G}_{\mathrm{EPLS}'}(p) = \begin{cases} \mathrm{EPS}_1 \text{ (see Example 4)} & \text{if } p = \mathcal{F}_{\mathrm{EPL}}, \\ \mathcal{G}_{\mathrm{EPLS}}(p) & \text{otherwise.} \end{cases}$
We have $\mathrm{EPLS}' \approxeq \mathrm{EPLS}$. Moreover, let $\mathrm{EPLS}'' = (\mathcal{M}_{\mathrm{EPL}}, \mathcal{G}_{\mathrm{EPLS}''})$, where $\mathcal{G}_{\mathrm{EPLS}''}(p) = \begin{cases} \mathrm{EPS}_2 \text{ (see Example 4)} & \text{if } p = \mathcal{F}_{\mathrm{EPL}}, \\ \mathcal{G}_{\mathrm{EPLS}}(p) & \text{otherwise.} \end{cases}$ We have $\mathrm{EPLS}'' \preceq \mathrm{EPLS}$ and $\mathrm{EPLS} \not\preceq \mathrm{EPLS}''$. Consider also the following four program signatures

$\mathrm{EPS}_{\mathrm{I}} =$
```
class Exp extends Object{
  String name;
}

class Lit extends Exp{
  Int val; Lit setLit(Int x);
}
```

$\mathrm{EPS}_{\mathrm{II}} =$
```
class Exp extends Object{
  String name;
  String toString();
}
class Lit extends Exp{
  Int val; Lit setLit(Int x);
}
```

$\mathrm{ESP}_{\mathrm{III}} =$
```
class Exp extends Object{
  String name;
}

class Lit extends Exp{
  Int val; Lit setLit(Int x);
}
class Add extends Exp {
  Exp a; Exp b;
}
```

$\mathrm{EPS}_{\mathrm{IV}} =$
```
class Exp extends Object{
  String name;
  String toString();
}
class Lit extends Exp{
  Int val; Lit setLit(Int x);
}
class Add extends Object{
 Exp a; Exp a;
}
```

and the SPLS $\mathrm{EPLS}_0 = (\mathcal{M}_{\mathrm{EPL}_0}, \mathcal{G}_{\mathrm{EPLS}_0})$ where $\mathcal{M}_{\mathrm{EPL}_0} = (\mathcal{F}_{\mathrm{EPL}_0}, \mathcal{P}_{\mathrm{EPL}_0})$ is as in Example 1 and $\mathcal{G}_{\mathrm{EPLS}_0}(p_i) = \mathrm{EPLS}_i$ for $i \in \{\mathrm{I}, \mathrm{II}, \mathrm{III}, \mathrm{IV}\}$. We have $\mathrm{EPLS}_0 \preceq \mathrm{EPLS}$ and $\mathrm{EPLS} \not\preceq \mathrm{EPLS}_0$.

Let $\mathrm{EPS}'_{\mathrm{I}}$, $\mathrm{EPS}'_{\mathrm{II}}$, $\mathrm{EPS}'_{\mathrm{III}}$, $\mathrm{EPS}'_{\mathrm{IV}}$ be the program signatures obtained from $\mathrm{EPS}_{\mathrm{I}}$, $\mathrm{EPS}_{\mathrm{II}}$, $\mathrm{EPS}_{\mathrm{III}}$, $\mathrm{EPS}_{\mathrm{IV}}$ by dropping all the fields (respectively), and let $\mathrm{EPLS}'_0 = (\mathcal{M}_{\mathrm{EPL}_0}, \mathcal{G}_{\mathrm{EPLS}'_0})$ be the SPLS such that $\mathcal{M}_{\mathrm{EPL}_0}$ is as above and $\mathcal{G}_{\mathrm{EPLS}'_0}(p_i) = \mathrm{EPLS}'_i$ for $i \in \{\mathrm{I}, \mathrm{II}, \mathrm{III}, \mathrm{IV}\}$. We have $\mathrm{EPLS}'_0 \preceq \mathrm{EPLS}_0$ and $\mathrm{EPLS}_0 \not\preceq \mathrm{EPLS}'_0$.

3 The Slice Operator for SPLSs of IFJ Programs

In this section we lift the feature model slice operator to SPLs in extensional form. In order to do this, we first introduce some auxiliary notions.

Given a feature model $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ and a set $\mathcal{F}_0$ of features, the slice $\Pi_{\mathcal{F}_0}(\mathcal{M}) = \mathcal{M}_0 = (\mathcal{F}_0, \mathcal{P}_0)$ determines a partition of $\mathcal{P}$. Namely, let $\mathbf{cpl}_{\mathcal{F}_0,\mathcal{M}} : \mathcal{P}_0 \to 2^{\mathcal{P}}$ be the function that maps each sliced product $p_0 \in \mathcal{P}_0$ to the set of products $\{p \mid p \in \mathcal{P} \text{ and } p_0 = p \cap \mathcal{F}_0\}$ that complete it, then:

1. $\mathbf{cpl}_{\mathcal{F}_0,\mathcal{M}}(p_0)$ is non-empty, for all $p_0 \in \mathcal{P}_0$;
2. $p' \neq p''$ implies $\mathbf{cpl}_{\mathcal{F}_0,\mathcal{M}}(p') \cap \mathbf{cpl}_{\mathcal{F}_0,\mathcal{M}}(p'') = \emptyset$, for all $p', p'' \in \mathcal{P}_0$; and
3. $\bigcup_{p \in \mathcal{P}_0} \mathbf{cpl}_{\mathcal{F}_0,\mathcal{M}}(p) = \mathcal{P}$.

The following definition introduces a canonical form for the elements of the equivalence classes of the relation $\approxeq$ between program signatures (introduced immediately after Definition 6).

Definition 10 (Thin program signatures). *We say that a program signature PS is in* thin form *(*thin *for short) to mean that, for all classes* $\mathtt{C}_1, \mathtt{C}_2 \in \mathrm{dom}(PS)$ *and for all attributes* $\mathtt{a} \in \mathrm{dom}(PS(\mathtt{C}_1))$*, if* $\mathtt{C}_2 <:_{PS} \mathtt{C}_1$ *then* $\mathtt{a} \notin \mathrm{dom}(PS(\mathtt{C}_2))$*. We denote* $\boldsymbol{thin}(PS)$ *the thin form of a program signature PS.*

Example 6 (Thin signature of the Expression Program). Recall the program EP and the signatures EPS and EPS_1 considered in Example 4, where $\mathrm{EPS} = \boldsymbol{signature}(\mathrm{EP})$. It is straightforward to check $\mathrm{EPS}_1 = \boldsymbol{thin}(\mathrm{EPS})$ holds.

Given a non-empty set of program signatures $\overline{PS} = PS_1, ..., PS_n$ $(n \geq 1)$ we write $\bigwedge \overline{PS}$ to denote the thin program signature that is the infimum (a.k.a. greatest lower bound) of $\overline{PS}$ with respect to the interface relation—it is a program signature (which is unique modulo program signature equivalence $\approxeq$) that exposes exactly the (classes, fields, methods and subtyping) declarations that are present in all the program signatures $\overline{PS}$. The following theorem states that $\bigwedge \overline{PS}$ is always defined.

Theorem 1 (Infimum for program signatures w.r.t. $\preceq$). *The thin program signature $\bigwedge \overline{PS}$ that is the infimum with respect to $\preceq$ of a non empty set of program signature $\overline{PS} = PS_1, ..., PS_n$ $(n \geq 1)$ is always defined.*

Proof. See Appendix A. □

The following definition lifts the feature model slice operator $\Pi_{\mathcal{F}_0}$ (Definition 2) to SPLSs.

Definition 11 (SPLS slice). *Let $\mathcal{F}_0$ be a set of features and, let $\mathtt{Z} = (\mathcal{M}_\mathtt{Z}, \mathcal{G}_\mathtt{Z})$ be an SPLS with feature model $\mathcal{M}_\mathtt{Z} = (\mathcal{F}_\mathtt{Z}, \mathcal{P}_\mathtt{Z})$ and generator $\mathcal{G}_\mathtt{Z}$. The slice operator $\Pi_{\mathcal{F}_0}$ on SPLSs returns the SPLS $\Pi_{\mathcal{F}_0}(\mathtt{Z}) = (\mathcal{M}_0, \mathcal{G}_0)$ where*

(i) $\mathcal{M}_0 = (\mathcal{F}_0, \mathcal{P}_0) = \Pi_{\mathcal{F}_0}(\mathcal{M}_\mathtt{Z})$*; and*
(ii) for each $p_0 \in \mathcal{P}_0$ we have that $\mathcal{G}_0(p_0) = \bigwedge_{p \in \mathbf{cpl}_{\mathcal{F}_0, \mathcal{M}_\mathtt{Z}}(p_0)} \mathcal{G}_\mathtt{Z}(p)$.

Note that not all the interfaces of an SPLS $\mathtt{Z}$ are slices of $\mathtt{Z}$ (cf. the observation immediately after Definition 3). Namely, $\Pi_{\mathcal{F}_0}(\mathtt{Z})$ is an interface (which is unique modulo SPLS equivalence $\approxeq$) that is the greatest (with respect to the $\preceq$ relation between SPLSs) interface of $\mathtt{Z}$ with exactly the features of $\mathtt{Z}$ that are in $\mathcal{F}_0$ I.e., if $\mathtt{Z}_1 \preceq \mathtt{Z}$ and $\mathtt{Z}_1$ has exactly the features of $\mathtt{Z}$ that are in $\mathcal{F}_0$, then $\mathtt{Z}_1 \preceq \Pi_{\mathcal{F}_0}(\mathtt{Z})$. The SPLS slice operator induces therefore a restriction of the SPLS interface operator by limiting the possibility to hide declarations from SPLS variants (cf. the second degree of freedom discussed immediately after Definition 9). Namely, if $\Pi_{\mathcal{F}_0}((\mathcal{F}_\mathtt{Z}, \mathcal{P}_\mathtt{Z}), \mathcal{G}_\mathtt{Z})) = ((\mathcal{M}_0, \mathcal{P}_0), \mathcal{G}_0)$ then, for all $p_0 \in \mathcal{P}_0$, we have that $\mathcal{G}_0(p_0)$ exposes exactly the declarations that are present in all the program signatures $\mathcal{G}_\mathtt{Z}(p)$ such that $p \in \mathbf{cpl}_{\mathcal{F}_0, (\mathcal{F}_\mathtt{Z}, \mathcal{P}_\mathtt{Z})}(p_0)$.

The SPL slice is an operator that given an SPL and a set of features returns its greatest interface with exactly the given features. It is defined as follows.

Definition 12 (SPL slice). *Given an SPL* $\mathtt{L}$ *and a set of features* $\mathcal{F}_0$ *we define* $\Pi_{\mathcal{F}_0}(\mathtt{L})$ *as* $\Pi_{\mathcal{F}_0}(\boldsymbol{signature}(\mathtt{L}))$.

Example 7 (Slicing the EPL). Consider the SPL EPL of Example 3 and the SPLSs EPLS, EPLS_0 and EPLS'_0 of Example 5. We have that EPLS_0 is a slice of both EPLS and EPL. Namely, $\text{EPLS}_0 = \Pi_{Y_0}(\text{EPLS}) = \Pi_{Y_0}(\text{EPL})$, where $Y_0 = \{\mathsf{Lit}, \mathsf{Add}, \mathsf{Print}\}$ are the features of EPLS_0. Instead, EPLS'_0 is not a slice of EPLS.

4 On Slicing Delta-oriented SPLSs of IFJ Programs

The extensional representation of SPLs allowed us to formulate notion of slice of an SPLS by abstracting from SPL implementation details. However, in order to investigate a practical slicing algorithm, we need to consider a representation of SPLs that reflects some implementation approach. To this aim, we first recall the propositional presentation of feature models (in Section 4.1) and the delta-oriented approach to implement SPLs, the definition delta-oriented SPL of IFJ programs, and the corresponding definition of SPLS (in Section 4.2). Then we illustrate the problem of devising a feasible algorithm for slicing delta-oriented SPLSs where the feature model is represented in propositional form (in Section 4.3).

4.1 Propositional Representation of Feature Models

The propositional representation of feature models works well in practice [38, 6, 47, 36]. In this representation, a feature model is given by a pair $(\mathcal{F}, \phi)$ where:

- $\mathcal{F}$ is a set of features, and
- ϕ is a propositional formula where the variables x are feature names: $\phi \ ::= \ x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \neg\phi$.

A propositional formula ϕ over a set of features $\mathcal{F}$ represents the feature models whose products are configurations $\{x_1, ..., x_n\} \subseteq \mathcal{F}$ $(n \geq 0)$ such that ϕ is satisfied by assigning value true to the variables x_i $(1 \leq i \leq n)$ and false to all other variables. More formally, given the propositional representation $\mathcal{M} = (\mathcal{F}, \phi)$ of a feature model, we denote $\mathcal{E}(\mathcal{M})$ its extensional representation, i.e, the feature model $(\mathcal{F}, \mathcal{E}(\phi))$ with $\mathcal{E}(\phi) = \{\, p \mid p \subseteq \mathcal{F} \text{ and } \mathcal{I}_p \models \phi \,\}$.

Example 8 (A proposition representation of the Expression Feature Model). Consider the feature model $\mathcal{M}_{\text{EPL}} = (\mathcal{F}_{\text{EPL}}, \mathcal{P}_{\text{EPL}})$ and the propositional formula ϕ_{EPL} introduced in Example 1. Then $(\mathcal{F}_{\text{EPL}}, \phi_{\text{EPL}})$ is a propositional representation of the feature model $\mathcal{M}_{\text{EPL}}$, i.e., $\mathcal{E}((\mathcal{F}_{\text{EPL}}, \phi_{\text{EPL}})) = \mathcal{M}_{\text{EPL}}$.

4.2 Delta-oriented SPLs and SPLSs

Delta-Oriented Programming (DOP) [40, 41], [3, Sect. 6.6.1] is a transformational approach to implement SPLs. The artifact base of a delta-oriented SPL consists

$AB ::= P\ \overline{DD}$		Artifact Base
$DD ::=$ **delta** $\mathtt{d}\{\overline{CO}\}$		Delta Declaration
$CO ::=$ **adds** CD $\mid$ **removes** $\mathtt{C}$ $\mid$ **modifies** $\mathtt{C}$[**extends** $\mathtt{C}'$]$\{\overline{AO}\}$		Class Operation
$AO ::=$ **adds** AD $\mid$ **removes** $\mathtt{a}$ $\mid$ **modifies** MD		Attribute Operation

Fig. 4: Syntax of IFΔJ SPL artifact base

of a *base program* (that might be empty) and of a set of *delta modules* (*deltas* for short). A delta is a container of program modifications (e.g., for IFJ programs, a delta can add, remove or modify classes). The configuration knowledge of a delta-oriented SPL associates to each delta an *activation condition* (determining the set of products for which that delta is activated) and specifies an *application ordering* between deltas: once a product is selected, the corresponding variant can be automatically generated by applying the activated deltas to the base program according to the application ordering. It is worth mentioning that the *Feature-Oriented Programming* (FOP) [5], [3, Sect. 6.1] approach to implement SPLs can be understood as the restriction of DOP where deltas correspond one-to-one to features and do not contain remove operations.

4.2.1 Delta-oriented SPLs of IFJ programs

Imperative Featherweight Delta Java (IFΔJ) [7] is a core calculus for delta-oriented SPLs of IFJ programs. The abstract syntax of the artifact base of an IFΔJ SPL is given in Figure 4. The artifact base comprises a (possibly empty) IFJ program P, and a set of deltas $\overline{DD}$. A delta declaration DD comprises the name d of the delta and class operations $\overline{CO}$ representing the transformations performed when the delta is applied to an IFJ program. A class operation can add, remove, or modify a class. A class can be modified by (possibly) changing its super class and performing attribute operations $\overline{AO}$ on its body. An attribute operation can add or remove fields and methods, and modify the implementation of a method by replacing its body. The new body may call the special method name `original`, which is implicitly bound to the previous implementation of the method.

Recall that, according to Convention 1, we assume that the deltas declared in an artifact base have distinct names, the class operations in each delta act on distinct classes, the attribute operations in each class operation act on distinct attributes, etc.

If the feature model of a delta-oriented SPL L is in propositional representation $(\mathcal{F}, \phi)$, then the configuration knowledge of L can be conveniently represented by a pair $\mathcal{K} = (\alpha, <)$ where:

- α (the *delta activation map*) is a function that associates to each delta d a propositional formula $\phi_{\mathtt{d}}$ such that $\phi \wedge \phi_{\mathtt{d}}$ represents the set of products that activate it; and

– $<$ (the *delta application order*) is a partial ordering between delta names.[6]

Therefore an IFΔJ SPL can be represented by a triple $\mathtt{L} = ((\mathcal{F}, \phi), AB, \mathcal{K})$.

The generator of $\mathtt{L}$, denoted by $\mathcal{G}_{\mathtt{L}}$, is a total function that associates each product p in $\mathcal{M}_{\mathtt{L}}$ with the IFJ program $\mathtt{d}_n(\cdots\ \mathtt{d}_1(P)\cdots)$, where P is the base program of $\mathtt{L}$ and $\mathtt{d}_1 \ldots, \mathtt{d}_n$ ($n \geq 0$) are the deltas of $\mathtt{L}$ activated by p (they are applied to P according to a total ordering that is compatible with the application order).[7]

In most presentation of delta-oriented SPLs (see, e.g, [40, 41]), the generator is considered to be a partial function in order to encompass ill-formed SPLs where, for some product, the generation of the associated variant fails. Recall that we focus on well-formed SPLs,[8] where generators are total functions and the generated products are well-typed IFJ programs—see [24, 13] for effective means to ensure the well-formedness of IFΔJ SPLs.

The extensional representation a delta-oriented SPL $\mathtt{L}$, denoted by $\mathcal{E}(\mathtt{L})$, is the SPL $(\mathcal{M}_{\mathtt{L}}, \mathcal{G}_{\mathtt{L}})$ where $\mathcal{M}_{\mathtt{L}}$ and $\mathcal{G}_{\mathtt{L}}$ are the feature model and the generator of $\mathtt{L}$, respectively.

4.2.2 Delta-oriented SPLSs of IFJ programs

A delta-oriented SPLS [19] can be understood as a delta-oriented SPL where the variants are program signatures. The abstract syntax of the artifact base of an IFΔJ SPLSs [19], *called artifact base signature,* is given in Figure 5. An artifact base signature ABS comprises a program signature PS and a set of *delta signatures* $\overline{DS}$ that are deltas deprived of method-modifies operations and method bodies.

If the feature model of a delta-oriented SPLS $\mathtt{Z}$ is in propositional representation $(\mathcal{F}, \phi)$, then the configuration knowledge of $\mathtt{Z}$ can be represented by a pair $\mathcal{K} = (\alpha, <)$ defined similarly to the configuration knowledge of a delta-oriented SPL. Therefore the IFΔJ SPLS can be represented by a triple $\mathtt{Z} = ((\mathcal{F}, \phi), ABS, \mathcal{K})$.

Also generator of a delta-oriented SPLS $\mathtt{Z}$, denoted by $\mathcal{G}_{\mathtt{Z}}$, and the extensional representation a delta-oriented SPLS $\mathtt{Z}$, denoted by $\mathcal{E}(\mathtt{Z})$, are defined as for delta-oriented SPLs.

Given two delta-oriented SPLSs $\mathtt{Z}_1$ and $\mathtt{Z}_2$ we say that:

– $\mathtt{Z}_1$ and $\mathtt{Z}_2$ are *extensional equivalent* to mean that their extensional representations are equivalent, i.e., $\mathcal{E}(\mathtt{Z}_1) \approxeq \mathcal{E}(\mathtt{Z}_2)$; and
– $\mathtt{Z}_1$ is an interface of $\mathtt{Z}_2$ (written $\mathtt{Z}_1 \preceq \mathtt{Z}_2$) to mean that $\mathcal{E}(\mathtt{Z}_1) \preceq \mathcal{E}(\mathtt{Z}_2)$.

The *signature of an IFΔJ SPL* $\mathtt{L}$, denoted as ***signature***$(\mathtt{L})$, is the SPLS obtained from $\mathtt{L}$ by dropping the method-modifies operations and the body of the methods

[6] As pointed out in [40, 41], the delta application order $<_{\mathtt{L}}$ is defined as a partial ordering to avoid over specification.

[7] We assume that all the total orders that are compatible with $<_{\mathtt{L}}$ yield the same generator—see [34, 7] for effective means to enforce this constraint.

[8] See footnote 4.

$$\begin{array}{lll} \mathit{ABS} & ::= \mathit{PS}\ \overline{\mathit{DS}} & \text{AB Signature} \\ \mathit{DS} & ::= \textbf{delta}\ \mathtt{d}\ \{\ \overline{\mathit{COS}}\ \} & \text{Delta Signature} \\ \mathit{COS} & ::= \textbf{adds}\ \mathit{CS}\ \mid\ \textbf{removes}\ \mathtt{C}\ \mid\ \textbf{modifies}\ \mathtt{C}\ [\textbf{extends}\ \mathtt{C}']\{\overline{\mathit{AOS}}\} & \text{CO Signature} \\ \mathit{AOS} & ::= \textbf{adds}\ \mathit{AS}\ \mid\ \textbf{removes}\ \mathtt{a} & \text{AO Signature} \end{array}$$

Fig. 5: Syntax of IFΔJ SPLS artifact base signature

in the artifact base. Note that the notion of signature of a delta-oriented SPL is consistent with the notion of signature defined for extensionally represented SPLs (introduced immediately after Definition 7). Namely, for all IFΔJ SPLs L we have that:

$$\mathcal{E}(\boldsymbol{signature}(\mathtt{L})) = \boldsymbol{signature}(\mathcal{E}(\mathtt{L})).$$

Given a delta-oriented SPLS Z and a delta-oriented SPL L, we say that Z is an interface of L (written $\mathtt{Z} \preceq \mathtt{L}$) to mean that $\mathcal{E}(\mathtt{Z}) \preceq \mathcal{E}(\boldsymbol{signature}(\mathtt{L}))$.

Recently [19], we have presented an algorithm for checking the interface relation between IFΔJ SPLSs where the feature model is represented in propositional form. The algorithm encodes interface checking into a boolean formula such that the formula is valid if and only of the interface relation holds. Then a SAT solver can be used to check whether a propositional formula is valid by checking whether its negation is unsatisfiable. Although this is a co-NP problem, similar translations into SAT constraints have been applied in practice for several SPL analysis with good results [26, 46, 47, 37].

4.3 On Devising an Algorithm for Slicing Delta-oriented SPLSs

Given a set of features $\mathcal{F}_0$ and delta-oriented SPL L where the feature model is represented in propositional form, manually writing a delta-oriented SPLS Z that is a slice of $\boldsymbol{signature}(\mathtt{L})$ for $\mathcal{F}_0$ is a tedious and error-prone task. In this section we illustrate the problem of devising a feasible algorithm for slicing delta-oriented SPLSs where the feature model is represented in propositional form.

We first focus on slicing a feature model represented in propositional form (in Section 4.3.1), then we consider slicing an IFΔJ SPLS (in Section 4.3.2).

4.3.1 Slicing Feature Models in Propositional Form

Given a set of features $X = \{x_1, ..., x_n\}$ ($n \geq 0$) and a feature model in propositional representation $(\mathcal{F}, \phi)$, the slicing algorithm ***slice*** is defined by:

$$\boldsymbol{slice}_X((\mathcal{F}, \phi)) = (\mathcal{F} \cap X, \boldsymbol{sliceBF}_{\mathcal{F}/X}(\phi))$$

where the algorithm ***sliceBF*** is defined by:

$$\begin{aligned} \boldsymbol{sliceBF}_{\emptyset}(\phi) &= \phi \\ \boldsymbol{sliceBF}_{\{x_1,\ldots,x_n\}}(\phi) &= \boldsymbol{sliceBF}_{\{x_2,\ldots,x_n\}}(\phi[x_1 := \textbf{true}]) \vee (\phi[x_1 := \textbf{false}])). \end{aligned}$$

The following theorem states that the slicing algorithm $\boldsymbol{slice}$ is correct.

Theorem 2 (Correctness of the *slice* algorithm for feature models). *For all set of features X and for all feature models in propositional representation $(\mathcal{F}, \phi)$, we have that $\mathcal{E}(\boldsymbol{slice}_X(\mathcal{F}, \phi))) = \Pi_X(\mathcal{E}((\mathcal{F}, \phi)))$.*

Proof. Straightforward by induction on the number of features in $\mathcal{F} \setminus X$. □

By construction, the size of feature model $\boldsymbol{slice}_X((\mathcal{F}, \phi))$ can grow as 2^n, where n is the number of variables in X. In order to avoid this exponential growth, we modify the notion of propositional representation of feature model (introduced in Section 4.1) by replacing the Boolean formula ϕ by an (existentially) Quantified-Boolean formula σ defined by:

$$\sigma \ ::= \ \exists \overline{x}.\phi, \quad \text{where } \overline{x} \text{ may be empty, i.e., } \sigma = \phi.$$

Given a set of features $X = \{x_1, ..., x_n\}$ $(n \geq 0)$ and a feature model in propositional representation $(\mathcal{F}, \sigma)$, the slicing algorithm $\boldsymbol{sliceE}$ is defined by:

$$\boldsymbol{sliceE}_X((\mathcal{F}, \exists \overline{y}.\phi)) = (\mathcal{F} \cap X, \exists \overline{w}.\phi)), \quad \text{where } \overline{w} \text{ are the elements of } \{\overline{y}\} \cup (\mathcal{F} \setminus X).$$

The following theorem states that the slicing algorithm $\boldsymbol{sliceE}$ is correct.

Theorem 3 (Correctness of the *sliceE* algorithm for feature models). *For all set of features X and for all feature models in propositional representation $(\mathcal{F}, \sigma)$, we have that $\mathcal{E}(\boldsymbol{sliceE}_X(\mathcal{F}, \sigma))) = \Pi_X(\mathcal{E}((\mathcal{F}, \sigma)))$.*

Proof. Straightforward by induction on the number of features in $\mathcal{F} \setminus X$. □

Example 9 (Computing a slice of the Expression Feature Model). Consider the feature model $(\mathcal{F}_{\mathrm{EPL}}, \phi_{\mathrm{EPL}})$ introduced in Example 8 and the set of features $Y_0 = \{\mathsf{Lit}, \mathsf{Add}, \mathsf{Print}\}$ introduced in Example 7. We have: $\boldsymbol{slice}_{Y_0}(\mathcal{F}_{\mathrm{EPL}}, \phi_{\mathrm{EPL}}) = (Y_0, \phi_{\mathrm{EPL}_0})$ and $\boldsymbol{sliceE}_{Y_0}(\mathcal{F}_{\mathrm{EPL}}, \phi_{\mathrm{EPL}}) = (Y_0, \sigma_{\mathrm{EPL}_0})$, where both $\phi_{\mathrm{EPL}_0} = (\mathsf{Lit} \wedge (\mathsf{Print} \vee \mathbf{true})) \vee (\mathsf{Lit} \wedge (\mathsf{Print} \vee \mathbf{false}))$ and $\sigma_{\mathrm{EPL}_0} = \exists x.\mathsf{Lit} \wedge (\mathsf{Print} \vee x)$ are logically equivalent to Lit.

Although the fact that slicing a feature model in propositional representation corresponds to performing an existential quantification on the dropped feature variables was already know in the literature (e.g., we have exploited it Section 5 of [19]), we are not aware of other authors that have published slicing algorithms like $\boldsymbol{slice}_X$ and $\boldsymbol{sliceE}_X$ above.

4.3.2 On the Problem of Slicing IF$\boldsymbol{\Delta}$J SPLSs

We aim at devising an algorithm such that:

- given an IFΔJ SPLS $\mathsf{Z} = ((\mathcal{F}, \sigma), ABS, \mathcal{K})$ and a set of features X,
- returns an IFΔJ SPLS $\mathsf{Z}' = ((\mathcal{F}', \sigma'), ABS', \mathcal{K}')$ that is a slice of Z for X and is such that:

1. $(\mathcal{F}', \sigma') = \textit{\textbf{sliceE}}_X(\mathcal{F}, \sigma)$,
2. the size of the artifact base ABS' is linear in the size of ABS, and
3. the size of the configuration knowledge $\mathcal{K}'$ is linear in the size of $\mathcal{K}$.

Note that requirements 2 and 3 above rule out any algorithm that returns an IFΔJ SPLS where the artifact base and configuration knowledge are the straightforward encoding of the generation mapping $\mathcal{G}_{\mathtt{Z}}$ of Definition 11 (i.e., a delta for each product, activated if and only if the product is selected).

Although we don't know whether an algorithm that satisfies requirements 2 and 3 exists, we conjecture that it exists at least for some significant classes of delta-oriented SPLs. We leave the investigating of such an algorithm for future work, and conclude this section by an example.

Example 10 (On slicing a delta-oriented implementation of the EPLS). Consider the IFΔJ SPLS $\mathtt{Z} = (\mathcal{M}_{\mathrm{EPL}}, ABS, \mathcal{K})$ where: $\mathcal{M}_{\mathrm{EPL}}$ is the feature model $(\mathcal{F}_{\mathrm{EPL}}, \phi_{\mathrm{EPL}})$ introduced in Example 8; the artifact base signatureABS is

```
EPS₁ // the program signature introduced in Example 4
delta d₁ { removes  Add }
delta d₂ { modifies Exp {removes toString} }   delta d₃ { modifies Exp {removes toInt} }
```

and the configuration knowledge $\mathcal{K}$ comprises the activation mapping $\{\mathtt{d}_1 \mapsto \neg\mathsf{Add},\ \mathtt{d}_2 \mapsto \neg\mathsf{Print},\ \mathtt{d}_3 \mapsto \neg\mathsf{Eval}\}$ and the flat application order (i.e., $\mathtt{d}_1$, $\mathtt{d}_2$ and $\mathtt{d}_3$ can be applied in any order). Note that $\mathcal{E}(\mathtt{Z}) \approxeq \mathrm{EPLS}_1$, where EPLS_1 is as in Example 5.

The slicing of $\mathtt{Z}$ w.r.t. the set of features $Y_0 = \{\mathsf{Lit}, \mathsf{Add}, \mathsf{Print}\}$ introduced in Example 5 is represented by the IFΔJ SPLS $\mathtt{Z}_0 = (\mathcal{M}_{\mathrm{EPL}_0}, ABS_0, \mathcal{K}_0)$ where: $\mathcal{M}_{\mathrm{EPL}_0}$ is the feature model (Y_0, Lit) introduced in Example 9; the artifact base signatureABS_0 is

```
EPS_IV // the program signature introduced in Example 5
delta d₁ { removes  Add }
delta d₂ { modifies Exp {removes toString} }
```

and the configuration knowledge $\mathcal{K}$ comprises the activation mapping $\{\mathtt{d}_1 \mapsto \neg\mathsf{Add},\ \mathtt{d}_2 \mapsto \neg\mathsf{Print}\}$ and the flat application order. Note that $\mathcal{E}(\mathtt{Z}_0) \approxeq \mathrm{EPLS}_0$, where EPLS_0 is as in Example 7.

5 Related Work

The notion of SPLS considered in this paper can be used to introduce a support for MPLs on top of a given approach for implementing SPL. For instance, in [19] we have exploited it to to define a formal model for delta-oriented MPLs. Previous work [25] informally outlined an extension of delta-oriented programming to implement MPLs, which does not enforce any boundaries between different SPLs and therefore is not suitable for supporting compositional analyses. In contrast, as illustrated in [19], SPLSs can be used to support compostional type-checking of MPLs of IFJ programs.

Schröter et al. [44] advocated investigating interface constructs for supporting compositional analyses of MPLs at different stages of the development process. In particular, they informally introduced the notion of syntactical interfaces (which generalizes feature model interfaces to provide a view of reusable programming artifacts) and the notion of behavioral interface (which generalizes syntactical interfaces to support formal verification). The notion of SPLS considered in this paper is (according to terminology of [44]) a syntactical interface.

Schröter et al. [45] also proposed the notion of feature-context interfaces in order to support preventing type errors while developing SPLs with the FOP approach. A feature-context interface provides an invariable API specifying classes and members of the feature modules that are intended to be accessible in the context of a given set of features. In contrast, an SPLS represents a variability-aware API.

The notion of slice of an SPLS for a set of features introduced and formalized in this paper lifts to SPLs the notion of slice of a feature model introduced in [1] (see also [43]). We are not aware of any other proposal for lifting to SPLs the notion of slice of a feature model.

6 Conclusions and Future Work

We have defined the notion of slice of an SPLS by abstracting from SPL(S) implementation approaches, and we have formulated the problem of defining an efficient algorithm that given a delta-oriented SPLS K and a set of features F returns a delta-oriented SPLS that is an slice of K for F.

In future work we would like to investigate a efficient algorithm for slicing delta-oriented SPLSs. In particular, we are planning to devise an algorithm for refactoring IFΔJ SPLSs to some normal form that is suitable for performing a slice. A starting point for this investigation could be represented by the algorithms for refactoring IFΔJ SPLs presented in [14, 15, 18]. The Abstract Behavioural Specification (ABS) language [9, 31, 16] is a delta-oriented modeling language has been successfully used in the context of industrial use cases [32, 28, 2, 11]. DeltaJava [33, 48] is a delta-oriented programming language designed to comfortably create SPLs within the Java environment. In future work we would like to exploit the notions of SPLS and slice for adding support for MPLs and support for deductive verification proof reuse [20, 8, 27] to the ABS toolchain (https://abs-models.org/) and to the DeltaJava toochain (http://deltajava.org/). We also plan the exploit the notion of SPLS to increase modularity in mechanisms that extend delta-oriented programming to support dynamic SPLs [23] (see also [21, 22]) and interoperability between variants of the same SPL [12].

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

A Proof of Theorem 1

Recall that, although `Object` $\notin$ dom(PS), class `Object` is used in every non-empty program PS . Therefore, $\leq:_{PS}$ is a relation on $\mathrm{dom}^{\circ}(PS)$, where $\mathrm{dom}^{\circ}(PS)$ is a shortening for dom(PS) $\cup$ {`Object`}.

Definition 13 (Subtyping path). *Given a program signature PS and a class* $\mathtt{C} \in \mathrm{dom}(PS)$*, we denote* $\textsc{path}(\mathtt{C}, PS)$ *the restriction of* $\leq:_{PS}$ *to the supertypes of* `C` *viz. the set* $\{(\mathtt{C}', \mathtt{C}'') \mid \mathtt{C}'{\leq:_{PS}}\mathtt{C}''$ *and* $\mathtt{C}{\leq:_{PS}}\mathtt{C}'\}$.

We remark that $\textsc{path}(\mathtt{C}, PS)$ is an order relation that identifies (uniquely) a linearly ordered sequence of classes, with `C` as bottom and `Object` as top. No path can be empty, since it has to include at least the pair (`Object`, `Object`).

The following definition and lemma exploit a canonical form for the elements of the equivalence classes of the relation $\approxeq$ between program signatures that is more convenient than the thin form (given in Definition 10) for writing the proofs.

Definition 14 (Fat program signatures *fatInf* operator). *We say that a program signature PS is* in fat form *(*fat *for short) to mean that, for all classes* $\mathtt{C} \in \mathrm{dom}(PS)$ *and for all attributes* $\mathtt{a} \in \mathrm{dom}(PS(\mathtt{C}))$*, if* $\mathit{lookup}_{PS}(\mathtt{a}, \mathtt{C}) = AS$ *then* $PS(\mathtt{C})(\mathtt{a}) = AS$*. We write* ***fat***$(PS)$ *to denote the fat form of a program signature PS. Let* $\overline{PS}$ *be a (non empty) set of program signatures.*

1. *We write* $\bigcap_{\overline{PS}} \mathrm{dom}(PS)$ *to shorten* $\bigcap_{PS\in\overline{PS}} \mathrm{dom}(PS)$*. Note that* `Object` *is never included in this intersection.*
2. *Let* $\mathtt{C} \in \bigcap_{\overline{PS}} \mathrm{dom}(PS)$.
   (a) $\textsc{path}_{\overline{PS}}(\mathtt{C})$ *is the linear order relation* $\bigcap\{\textsc{path}(\mathtt{C}, PS_i) \mid PS_i \in \overline{PS}\}$.
   (b) $\textsc{path}^{\not{\mathtt{C}}}_{\overline{PS}}(\mathtt{C})$ *is the order relation obtained by* $\textsc{path}_{\overline{PS}}(\mathtt{C})$ *removing* `C`.
   (c) *mcs*$(\mathtt{C})$ *(*minimum common superclass *of* `C`*) is the bottom of* $\textsc{path}^{\not{\mathtt{C}}}_{\overline{PS}}(\mathtt{C})$.
   (d) $\textsc{mcfd}(\overline{PS}, \mathtt{C})$ *is the (maximum) set of* common field declarations*, viz. the set of all field declarations of the shape* $\mathtt{C}_*\ \mathtt{f}_*$ *such that: for all* $PS_i \in \overline{PS}$, $\mathit{lookup}_{PS_i}(\mathtt{f}_*, \mathtt{C}) = \mathtt{C}_*\ \mathtt{f}_*$.
   (e) $\textsc{mcmd}(\overline{PS}, \mathtt{C})$ *is the (maximum) set of* common method declarations*, viz. the set of all field declarations of the shape* $\mathtt{C}_*\ \mathtt{m}_*(\overline{\mathtt{C}_{\mathtt{x}}\ \mathtt{x}})$ *such that: for all* $PS_i \in \overline{PS}$, $\mathit{lookup}_{PS_i}(\mathtt{m}_*, \mathtt{C}) = \mathtt{C}_*\ \mathtt{m}_*(\overline{\mathtt{C}_{\mathtt{x}}\ \mathtt{x}'})$ *for some variable names* $\mathtt{x}'$ *(the type sequences have to match but, as usual, the names of arguments do not matter).*
3. *We denote* ***fatInf***$(\overline{PS})$ *the in* $\overline{PS}$*, viz. the program signature such that, for all and only* $\mathtt{C} \in \bigcap_{\overline{PS}} \mathrm{dom}(PS)$ *includes all and only the declarations:*

$$\textsf{class}\ \mathtt{C}\ \textsf{extends}\ \mathit{mcs}(\mathtt{C})\ \{\ \textsc{mcmd}(\overline{PS}, \mathtt{C})\quad \textsc{mcfd}(\overline{PS}, \mathtt{C})\ \}\ .$$

Lemma 1 (*fatInf* characterizes $\bigwedge$). *For every (non empty) set of program signatures* $\overline{PS}$*, it holds that* ***fatInf***$(\overline{PS}) =$ ***fat***$(\bigwedge \overline{PS})$.

Proof. It is straightforward to see that $\boldsymbol{fatInf}(\overline{PS})$ is always defined and that $\boldsymbol{fatInf}(\overline{PS}) \preceq PS_i$ for all $PS_i \in \overline{PS}$ (since it is build as a restriction of them). Therefore $\boldsymbol{fatInf}(\overline{PS})$ is a lower bound for $\overline{PS}$ and we can conclude the proof by showing that it is the greater between the lower bounds for $\overline{PS}$, namely if $PS^\star \preceq PS_i$ for all $PS_i \in \overline{PS}$ then $PS^\star \preceq \boldsymbol{fatInf}(\overline{PS})$ has to hold. In accordance with Definition 5, we have to prove that the following three conditions hold.

(i) If $\mathtt{C} \in \mathrm{dom}(PS^\star)$ then it has to be $\mathtt{C} \in \mathrm{dom}(PS_i)$ for all $PS_i \in \overline{PS}$. Therefore, $\mathtt{C} \in \boldsymbol{fatInf}(\overline{PS})$ by construction.

(ii) Let $\mathtt{C}_1, \mathtt{C}_2 \in \mathrm{dom}(PS^\star)$. If $\mathtt{C}_0 <:_{P^\star} \mathtt{C}_1$ then it has to be $\mathtt{C}_0 <:_{P_i} \mathtt{C}_1$ for all $PS_i \in \overline{PS}$, viz. $(\mathtt{C}_0, \mathtt{C}_1) \in \textsc{path}_{\overline{PS}_i}(\mathtt{C}_0)$. Therefore, $(\mathtt{C}_0, \mathtt{C}_1) \in \textsc{path}_{(\overline{PS})}(\mathtt{C})$ by construction.

(iii) Let $\mathtt{C} \in \mathrm{dom}(PS^\star)$ and $\mathtt{a}$ be an attribute such that $\mathit{lookup}_{PS^\star}(\mathtt{a}, \mathtt{C})$ is defined. But $PS^\star \preceq \overline{PS}$ implies that, for all $PS_i \in \overline{PS}$, $\mathit{lookup}_{PS_i}(\mathtt{a}, \mathtt{C})$ is defined and $\mathit{lookup}_{PS_i}(\mathtt{a}, \mathtt{C}) = \mathit{lookup}_{PS^\star}(\mathtt{a}, \mathtt{C})$. Since $\textsc{mcfd}(\overline{PS}, \mathtt{C})$ and $\textsc{mcmd}(\overline{PS}, \mathtt{C})$ have been defined to grasp the maximum set of common attribute declarations, the proof follows by construction. □

Proof (of Theorem 1). Straightforward by Lemma 1. □

References

1. Acher, M., Collet, P., Lahire, P., France, R.B.: Slicing feature models. In: 26th IEEE/ACM International Conference on Automated Software Engineering, (ASE), 2011. pp. 424–427 (2011). https://doi.org/10.1109/ASE.2011.6100089
2. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. Service Oriented Computing and Applications **8**(4), 323–339 (2014). https://doi.org/10.1007/s11761-013-0148-0
3. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013)
4. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of International Software Product Line Conference (SPLC). LNCS, vol. 3714, pp. 7–20. Springer (2005). https://doi.org/10.1007/11554844_3
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering **30**, 355–371 (2004). https://doi.org/10.1109/TSE.2004.23
6. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6), 615–636 (2010). https://doi.org/10.1016/j.is.2010.01.001, https://doi.org/10.1016/j.is.2010.01.001
7. Bettini, L., Damiani, F., Schaefer, I.: Compositional type checking of delta-oriented software product lines. Acta Informatica **50**(2), 77–122 (2013). https://doi.org/10.1007/s00236-012-0173-z
8. Bubel, R., Damiani, F., Hähnle, R., Johnsen, E.B., Owe, O., Schaefer, I., Yu, I.C.: Proof repositories for compositional verification of evolving software systems - managing change when proving software correct. Transactions on Foundations for Mastering Change I **1**, 130–156 (2016). https://doi.org/10.1007/978-3-319-46508-1_8

9. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Formal Methods for Eternal Networked Software Systems, Lecture Notes in Computer Science, vol. 6659, pp. 417–457. Springer International Publishing (2011)
10. Clements, P., Northrop, L.: Software Product Lines: Practices & Patterns. Addison Wesley Longman (2001)
11. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M.: A unified and formal programming model for deltas and traits. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 424–441. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5_25
12. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M.: Interoperability of software product line variants. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1. p. 264–268. SPLC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3233027.3236401
13. Damiani, F., Lienhardt, M.: On type checking delta-oriented product lines. In: Integrated Formal Methods: 12th International Conference, iFM 2016. LNCS, vol. 9681, pp. 47–62. Springer (2016). https://doi.org/10.1007/978-3-319-33693-0_4
14. Damiani, F., Lienhardt, M.: Refactoring delta-oriented product lines to achieve monotonicity. In: Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016. EPTCS, vol. 206, pp. 2–16 (2016). https://doi.org/10.4204/EPTCS.206.2
15. Damiani, F., Lienhardt, M.: Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9953, pp. 579–596 (2016). https://doi.org/10.1007/978-3-319-47169-3_45
16. Damiani, F., Lienhardt, M., Muschevici, R., Schaefer, I.: An extension of the abs toolchain with a mechanism for type checking spls. In: Integrated Formal Methods. pp. 111–126. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_8
17. Damiani, F., Lienhardt, M., Paolini, L.: A formal model for multi SPLs. In: 7th International Conference on Fundamentals of Software Engineering (FSEN). Lecture Notes in Computer Science, vol. 10522, pp. 67–83. Springer, Berlin, Germany (2017). https://doi.org/10.1007/978-3-319-68972-2_5
18. Damiani, F., Lienhardt, M., Paolini, L.: Automatic refactoring of delta-oriented spls to remove-free form and replace-free form. Int. J. Softw. Tools Technol. Transf. **21**(6), 691–707 (2019). https://doi.org/10.1007/s10009-019-00534-2
19. Damiani, F., Lienhardt, M., Paolini, L.: A formal model for multi software product lines. Science of Computer Programming **172**, 203 – 231 (2019). https://doi.org/10.1016/j.scico.2018.11.005
20. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: Proc. of SPLC 2012 - Volume 2. pp. 53–60. ACM (2012). https://doi.org/10.1145/2364412.2364422

21. Damiani, F., Padovani, L., Schaefer, I.: A formal foundation for dynamic delta-oriented software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. pp. 1–10. GPCE '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2371401.2371403
22. Damiani, F., Padovani, L., Schaefer, I., Seidl, C.: A core calculus for dynamic delta-oriented programming. Acta Informatica **55**(4), 269–307 (Jun 2018). https://doi.org/10.1007/s00236-017-0293-6
23. Damiani, F., Schaefer, I.: Dynamic delta-oriented programming. In: Proceedings of the 15th International Software Product Line Conference, Volume 2. pp. 34:1–34:8. SPLC '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2019136.2019175
24. Damiani, F., Schaefer, I.: Family-based analysis of type safety for delta-oriented software product lines. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I. LNCS, vol. 7609, pp. 193–207. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_15
25. Damiani, F., Schaefer, I., Winkelmann, T.: Delta-oriented multi software product lines. In: Proceedings of the 18th International Software Product Line Conference - Volume 1. pp. 232–236. SPLC '14, ACM (2014). https://doi.org/10.1145/2648511.2648536
26. Delaware, B., Cook, W.R., Batory, D.: Fitting the pieces together: A machine-checked model of safe composition. In: ESEC/FSE. pp. 243–252. ACM (2009). https://doi.org/10.1145/1595696.1595733
27. Din, C.C., Johnsen, E.B., Owe, O., Yu, I.C.: A modular reasoning system using uninterpreted predicates for code reuse. J. Log. Algebraic Methods Program. **95**, 82–102 (2018). https://doi.org/10.1016/j.jlamp.2017.11.004
28. Helvensteijn, M., Muschevici, R., Wong, P.Y.H.: Delta modeling in practice: a Fredhopper case study. In: Proc. of VAMOS'12. pp. 139–148. ACM (2012). https://doi.org/10.1145/2110147.2110163
29. Holl, G., Grünbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. Information & Software Technology **54**(8), 828–852 (2012). https://doi.org/10.1016/j.infsof.2012.02.002
30. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS **23**(3), 396–450 (2001). https://doi.org/10.1145/503502.503505
31. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2012). https://doi.org/10.1007/978-3-642-25271-6_8
32. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Proc. of FTSCS 2016. CCIS, vol. 694, pp. 55–71. Springer (2017), dOI: 10.1007/978-3-319-53946-1_4
33. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: DeltaJ 1.5: delta-oriented programming for Java. In: International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14. pp. 63–74 (2014). https://doi.org/10.1145/2647508.2647512
34. Lienhardt, M., Clarke, D.: Conflict detection in delta-oriented programming. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I. pp. 178–192 (2012)

35. Lienhardt, M., Damiani, F., Donetti, S., Paolini, L.: Multi software product lines in the wild. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems. pp. 89–96. VAMOS 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3168365.3170425
36. Lienhardt, M., Damiani, F., Johnsen, E.B., Mauro, J.: Lazy product discovery in huge configuration spaces. In: Proceedings of the 42th International Conference on Software Engineering. ICSE '20, ACM (2020). https://doi.org/10.1145/3377811.3380372
37. Lienhardt, M., Damiani, F., Testa, L., Turin, G.: On checking delta-oriented product lines of statecharts. Science of Computer Programming **166**, 3 – 34 (2018). https://doi.org/j.scico.2018.05.007
38. Mendonca, M., Wasowski, A., Czarnecki, K.: SAT-based analysis of feature models is easy. In: Muthig, D., McGregor, J.D. (eds.) Proceedings of the 13th International Software Product Line Conference. ACM International Conference Proceeding Series, vol. 446, pp. 231–240. ACM (2009)
39. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005)
40. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Software Product Lines: Going Beyond (SPLC 2010). LNCS, vol. 6287, pp. 77–91 (2010). https://doi.org/10.1007/978-3-642-15579-6_6
41. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development. pp. 49–56. FOSD '10, ACM (2010). https://doi.org/10.1145/1868688.1868696
42. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer **14**(5), 477–495 (2012). https://doi.org/10.1007/s10009-012-0253-y
43. Schröter, R., Krieter, S., Thüm, T., Benduhn, F., Saake, G.: Feature-model interfaces: The highway to compositional analyses of highly-configurable systems. In: Proceedings of the 38th International Conference on Software Engineering. pp. 667–678. ICSE '16, ACM (2016). https://doi.org/10.1145/2884781.2884823
44. Schröter, R., Siegmund, N., Thüm, T.: Towards modular analysis of multi product lines. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops. pp. 96–99. SPLC'13, ACM (2013). https://doi.org/10.1145/2499777.2500719
45. Schröter, R., Siegmund, N., Thüm, T., Saake, G.: Feature-context interfaces: Tailored programming interfaces for spls. In: Proceedings of the 18th International Software Product Line Conference - Volume 1. pp. 102–111. SPLC'14, ACM (2014). https://doi.org/10.1145/2648511.2648522
46. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. pp. 95–104. GPCE '07, ACM (2007). https://doi.org/10.1145/1289971.1289989
47. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 6:1–6:45 (2014). https://doi.org/10.1145/2580950
48. Winkelmann, T., Koscielny, J., Seidl, C., Schuster, S., Damiani, F., Schaefer, I.: Parametric deltaj 1.5: Propagating feature attributes into implementation artifacts. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016. CEUR Workshop Proceedings, vol. 1559, pp. 40–54. CEUR-WS.org (2016)