A Formal Model for Multi Software Product Lines [☆]

Ferruccio Damiani[a], Michael Lienhardt[a], Luca Paolini[a]

[a]*Dipartimento di Informatica, University of Turin, Turin, Italy*

Abstract

A Software Product Line (SPL) is a family of similar programs generated from a common artifact base. A Multi SPL (MPL) is a set of interdependent SPLs that are typically managed and developed in a decentralized fashion. Delta-Oriented Programming (DOP) is a flexible and modular approach to implement SPLs. This paper presents new concepts that extend DOP to support the implementation of MPLs. These extensions aim to accommodate compositional analyses. They are presented by means of a core calculus for delta-oriented MPLs of Java programs. Suitability for MPL compositional analyses is demonstrated by compositional reuse of existing SPL analysis techniques.

Keywords: Core calculus, Delta-oriented programming, Featherweight Java, Multi software product line

1. Introduction

Highly-configurable software systems can be described as *Software Product Lines* (SPLs). An SPL is a family of similar programs, called *variants*, that have a well-documented variability and are generated from a common artifact base [1, 2, 3]. An SPL consists of: (i) a *feature model* defining the set of variants in terms of *features* (each feature represents an abstract descrip-

[☆]This work has been partially supported by: EU Horizon 2020 project HyVar (`www.hyvar-project.eu`), GA No. 644298; ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`); Ateneo/CSP D16D15000360005 project RunVar (`runvar-project.di.unito.it`).

Email addresses: `ferruccio.damiani@unito.it` (Ferruccio Damiani), `michael.lienhardt@unito.it` (Michael Lienhardt), `luca.paolini@unito.it` (Luca Paolini)

tion of functionality and each variant is identified by a set of features, called a *product*); (ii) an *artifact base* providing language dependent reusable code artifacts that are used to build the variants; and (iii) *configuration knowledge* which connects feature model and artifact base by defining how to derive variants from the code artifacts given the products (thus inducing a mapping from products to variants, called the *generator* of the SPL).

Delta-Oriented Programming (DOP) [4], [3, Sect. 6.6.1] is a modular approach to implement SPLs. The artifact base of a delta-oriented SPL consists of a *base program* (that might be empty) and of a set of *delta modules* (*deltas* for short), which are containers of program modifications (e.g., for Java programs, a delta can add, remove or modify classes and interfaces). The configuration knowledge of a delta-oriented SPL defines how the artifact base is used to generate the SPL's variants by associating to each delta an *activation condition* (i.e., a set of products for which that delta is activated) and specifying an *application ordering* between deltas: once a user selects a product, the corresponding variant is derived by applying the activated deltas to the base program according to the application ordering. Moreover, DOP is a generalization of *Feature-Oriented Programming* (FOP) [5], [3, Sect. 6.1], a previously proposed approach to implement SPLs where deltas correspond one-to-one to features and do not contain remove operations.

Modern software systems often out-grow the scale of SPLs by involving the notion of *Multi SPLs* (MPLs), i.e., sets of interdependent SPLs that need to be managed in a decentralized fashion by multiple teams and stakeholders [6]. There are two main motivations to build such MPLs: either to structure a complex SPL into more manageable modules, or to reuse existing SPLs into a bigger project.

An extension of DOP to implement MPLs has been informally outlined in [7] by proposing linguistic constructs for defining an MPL as an SPL that imports other SPLs. The feature model and the artifact base of the importing SPL is deeply integrated with the feature models and the artifact bases of the imported SPLs, respectively. This extension does not enforce any boundary between different SPLs—thus providing no support for compositional analyses.

In this paper we give, to the best of our knowledge, the first formal model of MPLs that spans feature model, artifact base and configuration knowledge. Our model is constructed around the concepts of *SPL signature*, *Dependent SPL* and *SPL composition*. It builds on recent work done by Schröter et al. [8] on compositional analysis of feature models, and on the delta-oriented programming core calculus IFΔJ by Bettini et al. [9], which is extended here to enable the construction of MPLs. The main achievement of our model is the ability to modularly compose and analyze SPLs by means of Dependent SPLs, which are SPLs with explicit dependencies, modeled by SPL signatures, that can be filled by SPLs (or Dependent SPLs) satisfying the given signatures.

This paper is an extended version of the prior [10] with: a more flexible notion of program interface (which, in turn, increases the flexibility of the induced notions of SPL interface and Dependent SPL interface); a more flexible notion of DPLs composition (that supports both composition of DPLs that share dependencies and partial composition); an improved formalization; a more thorough discussions; more explanations and examples; and the proofs of the main results.

Section 2 provides some background. Section 3 formalizes our notion of MPL by introducing the Imperative Featherweight Multi Delta Java (IFMΔJ) calculus, which extends IFΔJ with SPL signatures and Dependent SPLs. Section 4 defines the SPL composition mechanism that allow to build complex SPLs from simpler ones, and show that this mechanism supports compositionality of existing SPL analysis, like feature model analysis or type checking. Section 5 presents an algorithm to check the SPL interface relation. Section 6 discusses related work, and Section 7 concludes the paper by outlining possible directions for future work.

2. Background

2.1. IFΔJ: a formal foundation for delta-oriented SPLs

IFΔJ [9] is a core calculus for delta-oriented SPLs where variants are written in IFJ (an imperative version of FJ [11]). Figure 1 gives the abstract syntax of IFΔJ. Following [11], we use the overline notation for (possibly empty) sequences of elements: for instance $\overline{e}$ stands for a sequence of expressions $e_1, \ldots, e_n$ ($n \geq 0$); the empty sequence is denoted by $\emptyset$. Moreover, when no confusion may arise, we identify sequences of pairwise distinct elements with sets, e.g., we write $\overline{e}$ as short for $\{e_1, \ldots, e_n\}$.

2.1.1. IFJ programs

The abstract syntax of IFJ programs is given in Figure 1a. 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

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

(a) IFJ programs.

LD	$::=$ **line** $\mathtt{L}$ $\{\mathcal{M}\ \mathcal{K}\ AB\}$	SPL Declaration
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

(b) IFΔJ SPLs.

Figure 1: Syntax of IFΔJ.

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 class declaration *CD* we write dom(*CD*) to denote the set of attribute names declared in *CD*. Given a program *P*, a class name `C` and an attribute name `a`, we write dom(P), $<:_P$, $\mathtt{C}_P$ and $lookup_P(\mathtt{a},\mathtt{C})$ to denote, respectively: the set of class names declared in *P*; the subtyping relation in *P* (which is always supposed to be acyclic); the class declaration *CD* of `C` in *P* when it exists; and the declaration of the attribute `a` in the closest superclass of `C` (including `C` itself) that contains a declaration for `a` in *P*, when it exists.

Type system, operational semantics, and type soundness for IFJ are given in [9]. In the following, we say that an IFJ program is *well-typed* to mean that it can be typed by the typing rules given in [9].

2.1.2. IFΔJ SPLs

The abstract syntax of IFΔJ SPLs is given in Figure 1b. An SPL declaration comprises the name `L` of the product line, a feature model $\mathcal{M}$, a configuration knowledge $\mathcal{K}$, and an artifact base AB. The artifact base comprises a (possibly empty) IFJ program P, and a set of deltas $\overline{DD}$. A delta declaration DD comprises the name `d` of the delta and class operations $\overline{CO}$ representing the transformations performed when the delta is applied to an IFJ program. A class operation can add, remove, or modify a class. A class can be modified by (possibly) changing its super class and performing attribute operations $\overline{AO}$ on its body. An attribute operation can add or remove fields and methods, and modify the implementation of a method by replacing its body. The new body may call the special method name `original`, which is implicitly bound to the previous implementation of the method.

In order to ensure unambiguty in the artifact base, we require that the deltas in the artifact base must have distinct names, the class operations in a delta must act on distinct classes, and the attribute operations in a class operation must act on distinct attributes.

In IFΔJ there is no concrete syntax for the feature model and the configuration knowledge. As usual, to simplify the formalization, we represent feature models $\mathcal{M}$ as pairs (set of features, set of products) and configuration knowledges $\mathcal{K}$ as pairs (mapping from deltas to activation conditions, delta application ordering).

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

Definition 2 (Configuration knowledge). A *configuration knowledge* $\mathcal{K}_x$ is a pair $(\alpha_x, <_x)$ where α_x is a map that associates to each delta name the set of products that activate it ($\alpha_x(\mathtt{d})$ represents the *activation condition* of the delta `d`, i.e., a product p activates delta `d` if and only of $p \in \alpha_x(\mathtt{d})$), and $<_x$ is an ordering between delta names (the application ordering).

We assume *unambiguity* of the SPL, i.e., for each product, any total ordering of the activated deltas that respects the (possibly partial) order specified in $\mathcal{K}$ generates the same variant (we refer to [12, 9] for effective means to ensure unambiguity).

Feature model, configuration knowledge and artifact base of an SPL named `L` are denoted by $\mathcal{M}_{\mathtt{L}} = (\mathcal{F}_{\mathtt{L}}, \mathcal{P}_{\mathtt{L}})$, $\mathcal{K}_{\mathtt{L}} = (\alpha_{\mathtt{L}}, <_{\mathtt{L}})$ and $AB_{\mathtt{L}}$, respectively.

Remark 3 (On feature models and configuration knowledges). The representations described in Definitions 1 and 2 simplify stating and proving results independently from implementation details. However, they do not scale well in actual implementations. In the examples, we represent feature models also as *feature diagrams*, which are diagrams that illustrate feature dependencies by organizing features in a tree structure, possibly with *cross-tree constraints*, which are propositional formulas Φ where variables are feature names f:

$$\Phi ::= \mathbf{true} \ \mid \ f \ \mid \ \Phi \Rightarrow \Phi \ \mid \ \neg\Phi \ \mid \ \Phi \wedge \Phi \ \mid \ \Phi \vee \Phi$$

(see [13] for a discussion on other possible representations of feature models).

In the examples, we represent activation conditions as propositional formulas (see above) and application orderings as total orderings on a partition of the set of delta names.

Type system, variant generation, and type soundness for IFΔJ are given in [9]. We nonetheless recall in the following the variant generation process of IFΔJ and its notion of type safety, as they will be used in the rest of the document. In order to define the variant generation process in a Delta-Oriented SPL, we first introduce the auxiliary notions of delta applicability and delta application. A delta `d` is *applicable* to a program P iff each class to be added does not exist in P; each class to be removed or modified exists in P; and for every class-modify operation: in the class of P to be modified, each method or field to be added does not exist; each method or field to be removed exists; and each method to be modified exists and has the same header specified in the method-modify operation. If a delta `d` is applicable to P, then the *application* of `d` to P is the program, denoted by $\mathtt{d}(P)$, obtained from P by applying all the operations in `d`—otherwise $\mathtt{d}(P)$ is undefined.

Definition 4 (Generator of an SPL [9]). The *generator* of `L`, denoted by $\mathcal{G}_{\mathtt{L}}$, is the mapping that associates each product p in $\mathcal{M}_{\mathtt{L}}$ to the IFJ program $\mathtt{d}_n(\cdots\ \mathtt{d}_1(P)\cdots)$, where P is the base program of `L` and $\mathtt{d}_1 \ldots, \mathtt{d}_n$ ($n \geq 0$) are the deltas of `L` activated by p, listed according to the application order.

The generator $\mathcal{G}_{\mathtt{L}}$ may be partial since, for some product of `L`, a delta $\mathtt{d}_i$ $(1 \leq i \leq n)$ may not be applicable to the intermediate variant $\mathtt{d}_{i-1}(\cdots \mathtt{d}_1(P)\cdots)$ thus making $\mathcal{G}_{\mathtt{L}}$ undefined for that product. We write $\mathrm{dom}(\mathcal{G}_{\mathtt{L}})$ to denote the set of products for which $\mathcal{G}_{\mathtt{L}}$ is defined.

Definition 5 (Type safe IFΔJ SPL [9]). An IFΔJ SPL `L` is *type safe* iff the generator $\mathcal{G}_{\mathtt{L}}$ is total and all the variants are well-typed IFJ programs.

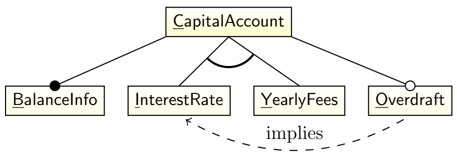
Example 1 (CapitalAccount SPL). Figure 2 illustrates the expressivness of the IFΔJ language by describing an SPL of capital accounts.[1] The SPL, named CapitalAccount, provides a class `CapAccount` for money managing bank accounts. The mandatory feature BalanceInfo provides some basic fields (`identity`, `balance` and `lastUpdate`) and a method `withdraw` (method `deposit`, which is similar, is omitted). Features InterestRate and YearlyFees provide two alternative bank-policies: one and only one of them, must be selected. The former manages accrued interests and operation-fees (applied to each withdraw), the second manages fixed fees per year (and no bank interests). The optional feature Overdraft allows to withdraw more money than that available, and requires feature InterestRate in order to apply a negative interest.

2.2. Feature model composition and interfaces

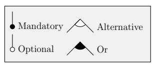
Recently, Schröter et al. [8] considered a notion of feature model composition through aggregation (i.e., by inclusion of one feature model into another feature model [14]) and proposed to use it in combination with a notion of feature model interface in order to support compositional analyses of feature models.

Definition 6 (Feature model composition [8]). Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, and $\mathcal{M}_{Glue} = (\mathcal{F}_{Glue}, \mathcal{P}_{Glue})$ be feature models that satisfy the *glue-proviso* $\mathcal{F}_{Glue} \subseteq \mathcal{F}_x \cup \mathcal{F}_y$. The *composition* of $\mathcal{M}_x$ and $\mathcal{M}_y$ is the feature model, denoted as $\mathcal{M}_{x/y}$, defined as follows by using *composition*

[1]To improve readability, in the artifact bases of the SPLs considered in this document's examples, we use Java syntax for field initialization, primitive data types, strings and sequential composition—encoding in IFΔJ syntax is straightforward (see [9]).



$$\begin{array}{lccccccc}
\mathcal{F}_{\textsf{CapitalAccount}} = & \{ & \textsf{C} & \textsf{B} & \textsf{I} & \textsf{Y} & \textsf{O} & \} \\
\mathcal{P}_{\textsf{CapitalAccount}} = & \{\{ & \textsf{C} & \textsf{B} & \textsf{I} & & & \}, \\
 & \{ & \textsf{C} & \textsf{B} & & \textsf{Y} & & \}, \\
 & \{ & \textsf{C} & \textsf{B} & \textsf{I} & & \textsf{O} & \}\,\}
\end{array}$$



$<_{\textsf{CapitalAccount}}$: $\{\texttt{dInterest}, \texttt{dFixFees}\} < \{\texttt{dOverdraft}\}$
$\alpha_{\textsf{CapitalAccount}}$: $\texttt{dInterest} \mapsto \textsf{I}, \texttt{dFixFees} \mapsto \textsf{Y}, \texttt{dOverdraft} \mapsto \textsf{O}$

```
class CapAccount extends Object {                                        // Base Program
        String identity;
        double balance = 0.0;
        Date lastUpdate=new Date().today();
        void withdraw(double x){ if (x>0) balance = balance−x; }
}

delta dInterest {                                                         // Deltas
      modifies class CapAccount {
              adds double yearRate=0.05;
              adds double opFees=1;
              adds void interestUpdate(double rate) { double range = lastUpdate.daysSince()/365;
                                                      lastUpdate = new Date().today();
                                                      balance += balance∗rate∗range; }
              modifies void withdraw(double x) { interestUpdate(yearRate);
                                                 if (x+opFees<=balance) original(x+opFees); }
      }
}
delta dFixFees {
      modifies class CapAccount {
              adds double yearFees = 10.0;
              adds Date yearPaid = new Date().currentYear();
              modifies void withdraw(double x) { balance −= yearFees∗(yearPaid.yearsSince());
                                                 yearPaid = new Date().currentYear();
                                                 if (x<=balance) original(x); }
      }
}
delta dOverdraft {
      modifies class CapAccount {
              adds double maxOver=100.0;
              adds double negativeRate=0.10;
              adds void negUpdate() { if (balance<0) interestUpdate(−negativeRate);
                                      else interestUpdate(yearRate); }
              modifies void withdraw(double x) { negUpdate();
                                                 balance+=maxOver;
                                                 original(x);
                                                 balance−=maxOver; }
      }
}
```

Figure 2: CapitalAccount SPL: feature model $\mathcal{M}_{\textsf{CapitalAccount}}$ (top), configuration knowledge $\mathcal{K}_{\textsf{CapitalAccount}}$ (middle), and artifact base $AB_{\textsf{CapitalAccount}}$ (bottom).

operation $\circ$, the auxiliary *join* operation $\bullet$, and the auxiliary operation $\mathcal{R}$:

$$\begin{aligned}
\mathcal{M}_{x/y} &= \circ(\mathcal{M}_x, \mathcal{M}_y, \mathcal{M}_{Glue}) = \mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_y = (\mathcal{M}_x \bullet \mathcal{R}(\mathcal{M}_y)) \bullet \mathcal{M}_{Glue} \\
\mathcal{R}(\mathcal{M}_y) &= (\mathcal{F}_y, \mathcal{P}_y \cup \{\emptyset\}) \\
\mathcal{M}_x \bullet \mathcal{M}_y &= (\mathcal{F}_x \cup \mathcal{F}_y, \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\})
\end{aligned}$$

Operation $\mathcal{R}$ takes one feature model $\mathcal{M}_y$ as input and converts it to a new feature model in which the empty product is a valid product. Operation $\bullet$ is similar to a cross product from relational algebra and creates all combinations between both product sets—it is associative and commutative, with $\mathcal{M}_{Id} = \mathcal{R}(\mathcal{M}_\emptyset) = \mathcal{R}((\emptyset, \emptyset)) = (\emptyset, \{\emptyset\})$ as identity (the proof is given in Appendix A).

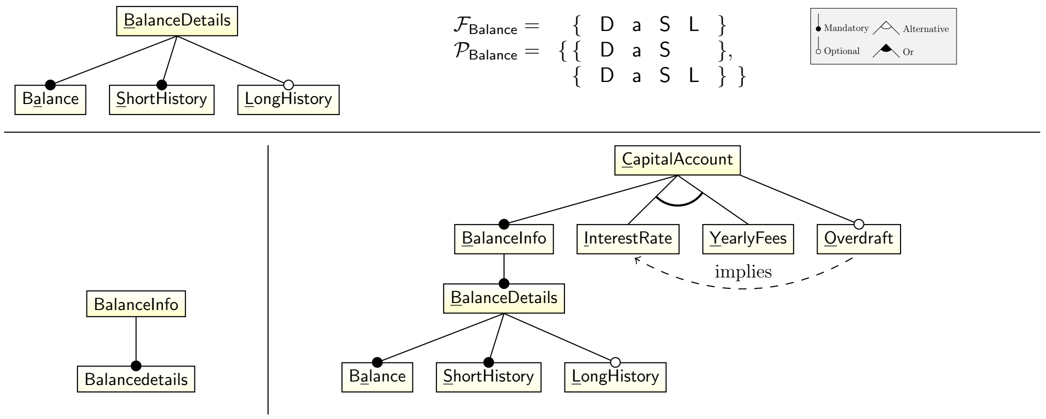
Intuitively, the composition $\mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_y$ corresponds to extending the feature model $\mathcal{M}_x$ with $\mathcal{M}_y$, where $\mathcal{M}_{Glue}$ describes inter-model constraints between $\mathcal{M}_x$ and $\mathcal{M}_y$, effectively specifying how $\mathcal{M}_y$ extends $\mathcal{M}_x$.

Example 2 (Feature model composition). Consider the feature model of the CapitalAccount SPL, presented in Figure 2, and imagine that one wants to refine its BalanceInfo feature as described in the feature model $\mathcal{M}_{\mathsf{Balance}}$ presented in Figure 3 (top). This refinement can be expressed as a feature model composition, where $\mathcal{M}_{\mathsf{CapitalAccount}}$ is extended with $\mathcal{M}_{\mathsf{Balance}}$, with the constraint that the feature BalanceInfo in $\mathcal{M}_{\mathsf{CapitalAccount}}$ is equivalent to the feature BalanceDetails in $\mathcal{M}_{\mathsf{Balance}}$. This constraint can easily be expressed with a feature model $\mathcal{F}_{Glue}$ with two features, BalanceInfo and BalanceDetails, both being mandatory, as shown in Figure 3 (bottom left). The feature model $\mathcal{M}_{\mathsf{CapitalAccount/Balance}} = \mathcal{M}_{\mathsf{CapitalAccount}} \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{\mathsf{Balance}}$ resulting of the composition, where the feature BalanceInfo is refined with the feature model $\mathcal{M}_{\mathsf{Balance}}$, is presented in Figure 3 (bottom right).

Definition 7 (Interface relation for feature models [8]). A feature model $\mathcal{M}_{Int} = (\mathcal{F}_{Int}, \mathcal{P}_{Int})$ is an *interface* of feature model $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, denoted as $\mathcal{M}_{Int} \preceq \mathcal{M}_x$, iff $\mathcal{F}_{Int} \subseteq \mathcal{F}_x$ and $\mathcal{P}_{Int} = \{p \cap \mathcal{F}_{Int} \mid p \in \mathcal{P}_x\}$.

Intuitively, an interface of a feature model $\mathcal{M}_x$ is simply the result of hiding some features from $\mathcal{M}_x$, like an API that only exposes public functionalities and hide internal helper functions.

Note that the interface relation for feature models is a partial order (i.e., it is reflexive, transitive and anti-symmetric). Moreover, given the feature model $\mathcal{M}_x$ and the set of features $\mathcal{F}_{Int} \subseteq \mathcal{F}_x$, the feature model $\mathcal{M}_{Int}$ is completely determined.



$$\mathcal{F}_{\mathsf{Balance}} = \{\ \mathsf{D}\ \ \mathsf{a}\ \ \mathsf{S}\ \ \mathsf{L}\ \}$$
$$\mathcal{P}_{\mathsf{Balance}} = \{\{\ \mathsf{D}\ \ \mathsf{a}\ \ \mathsf{S}\ \},\ \{\ \mathsf{D}\ \ \mathsf{a}\ \ \mathsf{S}\ \ \mathsf{L}\ \}\}$$

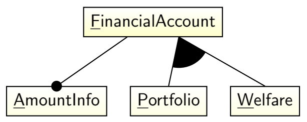
Figure 3: Example of Feature Model Composition: feature model $\mathcal{M}_{\mathsf{Balance}}$ (top), glue feature model $\mathcal{F}_{Glue}$ (bottom left) and resulting feature model $\mathcal{M}_{\mathsf{CapitalAccount/Balance}} = \mathcal{M}_{\mathsf{CapitalAccount}} \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{\mathsf{Balance}}$ (bottom right).

3. IFMΔJ: a core calculus for MPLs

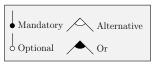
To introduce our MPL model, imagine another account SPL was developped concurrently to the one presented in Example 1.

Example 3 (FinancialAccount SPL). The FinancialAccount SPL, presented in Figure 4, has different characteristics from the CapitalAccount: it provides a class `FinAccount` for investment product managing bank accounts; its mandatory feature `AmountInfo` provides basic fields (identity, liquidity) for that class; and at least one feature between `Portfolio` and `Welfare` must be selected. The latter provides a list of welfare products while the former provides a list of financial products.

The SPLs CapitalAccount and FinancialAccount thus describe two kinds of bank accounts, and it would make perfect sense to combine them to obtain a new SPL describing a bank account including all the functionalities implemented in the two SPLs. A simple and direct approach would be to define a composition operator between SPLs, and construct this new SPL, called DualAccount, as a combination of the two CapitalAccount and FinancialAccount SPLs. However, this solution is not safisfactory as it would couple too strongly DualAccount to the SPLs that compose it: if a more convenient implementation of CapitalAccount or FinancialAccount comes up, it can be difficult to update the definition of DualAccount to use it. This is a situation similar to what can happen in Java-like programs: using only *classes* to



$$\begin{array}{rl}\mathcal{F}_{\mathsf{FinancialAccount}} = & \{\ \mathsf{F}\ \ \mathsf{A}\ \ \mathsf{P}\ \ \mathsf{W}\ \} \\ \mathcal{P}_{\mathsf{FinancialAccount}} = & \{\{\ \mathsf{F}\ \ \mathsf{A}\ \ \mathsf{P}\ \}, \\ & \ \ \{\ \mathsf{F}\ \ \mathsf{A}\ \ \mathsf{W}\ \}, \\ & \ \ \{\ \mathsf{F}\ \ \mathsf{A}\ \ \mathsf{P}\ \ \mathsf{W}\ \}\ \}\end{array}$$



$<_{\mathsf{FinancialAccount}}$: $\{\texttt{dPortfolio}, \texttt{dWelfare}\}$
$\alpha_{\mathsf{FinancialAccount}}$: $\texttt{dPortfolio} \mapsto \mathsf{P}, \texttt{dWelfare} \mapsto \mathsf{W}$

```
class FinAccount extends Object {                                          // Base program
        String identity;
        double liquidity=0.0;
}

delta dPortfolio {                                                         // Deltas
      adds class RiskProd extends Object {
                    String info;
                    int quantity;
                    RiskProd init(String i, int q){info=i; quantity=q; return this;}
      }
      modifies class FinAccount {
                adds LinkedList portfolio=new LinkedList();
                adds void addToPortfolio(String i, int q){ portfolio.add(new RiskProd().init(i,q));}
      }
}
delta dWelfare {
      adds class LifeProd extends Object {
                    String info;
                    String beneficiary;
                    LifeProd init(String i, String b){ info=i; beneficiary=b; return this; }
      }
      modifies class FinAccount {
                adds ArrayList welfare=new ArrayList();
                adds void addToWelfare(String i, String b){ welfare.add(new LifeProd().init(i,b)); }
      }
}
```

Figure 4: FinancialAccount SPL: feature model $\mathcal{M}_{\mathsf{FinancialAccount}}$ (top), configuration knowledge $\mathcal{K}_{\mathsf{FinancialAccount}}$ (middle), and artifact base$AB_{\mathsf{FinancialAccount}}$ (bottom).

specify the type of an object is very restrictive, and using *interface* instead can really help make a program more flexible and maintainable.

To overcome this issue, we introduce the notions of *SPL signature* and *Dependent SPL*: in our example, DualAccount becomes a Dependent SPL with two dependencies, one for each type of account. Each these dependencies is described by an SPL signature that specifies an API on which DualAccount depends, so that any SPL that implements such signature can fulfill that dependency. Hence, our approach to define the DualAccount Dependent SPL follows the structure presented below.

$PS ::= \overline{CS}$ Program Signature

$CS ::= \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{C}\ \{\ \overline{AS}\ \}$ Class Signature

$AS ::= FD \mid MH$ Attribute (Field or Method) Signature

(a) Program Signatures.

$LS ::= \textbf{sig}\ \mathtt{Z}\ \{\mathcal{M}\ \mathcal{K}\ ABS\}$ SPL Signature Declaration

$ABS ::= PS\ \overline{DS}$ AB Signature

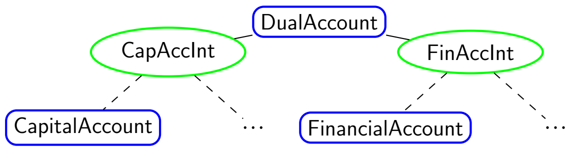
$DS ::= \textbf{delta}\ \mathtt{d}\ \{\ \overline{COS}\ \}$ Delta Signature

$COS ::= \textbf{adds}\ CS \mid \textbf{removes}\ \mathtt{C} \mid \textbf{modifies}\ \mathtt{C}\ [\textbf{extends}\ \mathtt{C}']\{\overline{AOS}\}$ CO Signature

$AOS ::= \textbf{adds}\ AS \mid \textbf{removes}\ \mathtt{a}$ AO Signature

(b) SPL Signatures.

Figure 5: Syntax of SPLS.



DualAccount depends on two SPL signatures: CapAccInt specifies the API requested by DualAccount for the capital account backend implementation, while FinAccInt specifies the API requested by DualAccount for the financial account implementation. Then these two signatures are implemented by CapitalAccount and FinancialAccount respectively, and possibly other SPLs.

We structure the presentation of our model as follows: in Section 3.1 we introduce the notion of SPL signature (SPLS); in Section 3.2 we formally define when an SPL implements an SPLS; and in Section 3.3 we define the notions of Dependent SPL (DPL) and of MPL.

3.1. Program signatures and SPL signatures

As previously discussed, an *SPL signature* (SPLS) describes the API of an SPL, i.e., it describes the APIs of the variants generated by that SPL. Since the different variants of an SPL can have a different API, we structure an SPLS as an SPL over API variants. Like a normal SPL, an SPLS is structured with a feature model, a configuration knowledge, and an artifact base. Its difference with an SPL lies in the fact that its artifact base is constructed from an IFJ program API (or *signature*) that does not include the implementation of methods, and deltas that manipulate such signature.

3.1.1. Program signatures

The abstract syntax of program signatures is given in Figure 5a. A program signature is a program deprived of method bodies, and a class signature is a class deprived of method bodies.

Given a class signature CS we write $\mathrm{dom}(CS)$ to denote the set of attribute names declared in CS. Given a program signature PS, a class name `C` and an attribute name `a`, we write $\mathrm{dom}(PS)$, $<:_{PS}$, $\mathtt{C}_{PS}$ and $lookup_{PS}(\mathtt{a}, \mathtt{C})$ to denote, respectively: the set of class names declared in PS; the subtyping relation in PS; the class signature CS of `C` in PS when it exists; and the signature of the attribute `a` in the closest supertype of `C` (including itself) that contains a declaration for `a` in PS, when it exists.

The *signature of a program* P, denoted as $\textbf{\textit{signature}}(P)$, is the program signature obtained from P by dropping the body of the methods. The *stub-completion of a program signature* PS, written $PS^{\bigstar}$, is the program obtained by adding the body {**return null;**} to all the method declarations in PS.

3.1.2. SPL signatures

The abstract syntax of SPLSs is given in Figure 5b. An SPLS declaration LS comprises the name `Z` of the SPLS, a feature model $\mathcal{M}$, a configuration knowledge $\mathcal{K}$, and an *artifact base signature* ABS which, in turn, comprises a program signature PS and a set of *delta signatures* $\overline{DS}$ that are deltas deprived of method-modifies operations and method bodies.

Like with SPLs, the feature model, configuration knowledge and artifact base of an SPLS named `Z` are denoted by $\mathcal{M}_{\mathtt{Z}} = (\mathcal{F}_{\mathtt{Z}}, \mathcal{P}_{\mathtt{Z}})$, $\mathcal{K}_{\mathtt{Z}} = (\alpha_{\mathtt{Z}}, <_{\mathtt{Z}})$ and $AB_{\mathtt{Z}}$, respectively.

The *signature of an SPL* `L`, denoted as $\textbf{\textit{signature}}(\mathtt{L})$, is the SPLS obtained from `L` by dropping the body of the methods in the artifact base. The *stub-completion of an SPLS* `Z`, written $\mathtt{Z}^{\bigstar}$, is the SPL obtained by adding the body {**return null;**} to all the method declarations in `Z`.

As an SPLS has the same structure of an SPL, we can associate to it a semantics that generates variants from products, where the variants are program signatures instead of programs:

Definition 8 (Generator of an SPLS). The *generator* of an SPLS `Z`, denoted by $\mathcal{G}_{\mathtt{Z}}$, is a mapping from products to program signatures defined similarly to the generator of an SPL (see Definition 4).

3.2. Interface relation for SPLs

We construct the *interface* relation, that states when an SPL implements an SPLS, in several steps. In Section 3.2.1, we define the *program subsignature* relation (that captures the fact that a program signature expresses a subset of the requirements expressed by another) and the *program interface* relation (defining when a program implements a signature). Then, in Section 3.2.2, we lift these relations to the SPL level, by defining the *SPL subsignature* relation (capturing the fact that an SPLS expresses a subset of the requirements expressed by another) and the *SPL interface* relations (defining when an SPLS implements an SPL).

3.2.1. Subsignature and interface relations for programs

A program signature (see Figure 5a) describes a program API and is used to express requirements on programs. The following subsignature relation formalizes the fact that a signature PS_1 expresses less requirement than another signature PS_2, while the interface relation formalizes the fact that a program signature PS is implemented by a program P (i.e., that all the classes and attributes declared in PS are defined in P, and $\mathtt{C}_1 {<:_{PS}} \mathtt{C}_2$ implies $\mathtt{C}_1 {<:_{P}} \mathtt{C}_2$).

Definition 9 (Subsignature relation on program signatures). A program signature PS_1 is a *subsignature* of a program signature PS_2, denoted as $PS_1 \preceq PS_2$, iff: (i) $\mathrm{dom}(PS_1) \subseteq \mathrm{dom}(PS_2)$; (ii) $<:_{PS_1} \subseteq <:_{PS_2}$; and, (iii) for all class name $\mathtt{C} \in \mathrm{dom}(PS_1)$ and all attribute name $\mathtt{a} \in \mathrm{dom}(\mathtt{C}_{PS_1})$, we have that $\mathit{lookup}_{PS_2}(\mathtt{a}, \mathtt{C})$ is defined and $\mathit{lookup}_{PS_1}(\mathtt{a}, \mathtt{C}) = \mathit{lookup}_{PS_2}(\mathtt{a}, \mathtt{C})$.[2]
(Interface relation for programs). A program signature PS is an *interface* of program P (or, equivalently, program P *implements* the program signature PS), denoted as $PS \preceq P$, iff $PS \preceq \boldsymbol{signature}(P)$ holds.

Note that the subsignature relation on program signatures is reflexive and transitive. However, due to the possibility of inheritance and overriding of

[2]Note that [10] defines this subsignature relation more simply with (i) the subtyping relation of PS_1 being included in PS_2 (as in here), and with the definition of each class in PS_1 being included in its corresponding definition in PS_2. The current definition, which has been introduced in [15] for a extension of IFJ with packages, is based on the *lookup* function instead of the actual definition of a class. It is more flexible than the definition in [10] as it allows to move fields and methods from a class to any of its descendants.

methods, it is not anti-symmetric (i.e., $PS_1 \preceq PS_2$ and $PS_2 \preceq PS_1$ do not imply $PS_1 = PS_2$).

Example 4 (Interface of a Capital Account). Consider the following variant P of the CapitalAccount SPL, generated from the product {BalanceInfo, `YearlyFees`} (see Example 1):

```
class CapAccount extends Object {
        String identity;
        double balance = 0.0;
        double yearFees = 10.0;
        Date lastUpdate=new Date().today();
        Date yearPaid = new Date().currentYear();
        void withdraw(double x){ balance −= yearFees∗(yearPaid.yearsSince());
                                   yearPaid = new Date().currentYear();
                                   if (x<=balance) { if (x>0) balance = balance−x; }
                                 }
}
```

The signature ***signature***(P) of this program is simply obtained by removing all definitions (field assignments and method definition) from P, and it is easy to see that the program signature PS defined as follows is a valid interface for P, as everything it declares is contained in ***signature***(P):

```
signature(P) =                              PS =
class CapAccount extends Object {           class CapAccount extends Object {
        String identity;                            String identity;
        double balance;                             double balance;
        double yearFees;                            Date lastUpdate;
        Date lastUpdate;                            void withdraw(double x);
        Date yearPaid;                      }
        void withdraw(double x);
}
```

3.2.2. Subsignature and interface relations for SPLs

The following subsignature relation on SPLSs naturally extends the one on program signatures (given in Definition 9) with the notion of feature model interface introduced in [8] (see Definition 7), while the interface relation formalizes when an SPL `L` implements an SPLS `Z`.

Definition 10 (Subsignature relation on SPLSs). An SPLS $\mathtt{Z}_1$ is a *subsignature* 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) the generators $\mathcal{G}_{\mathtt{Z}_1}$ and $\mathcal{G}_{\mathtt{Z}_2}$ are total and for each $p \in \mathcal{P}_{\mathtt{Z}_2}$, $\mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1}) \preceq \mathcal{G}_{\mathtt{Z}_2}(p)$.
(Interface relation for SPLs). An SPLS `Z` is an *interface* of an SPL `L` (or, equivalently, SPL `L` *implements* SPLS `Z`), denoted as $\mathtt{Z} \preceq \mathtt{L}$, iff $\mathtt{Z} \preceq$ ***signature***$(\mathtt{L})$ holds.

The subsignature relation for SPLSs has two degrees of freedom: it allows to hide features from the feature model (as described in Definition 7), and it allows to hide declarations from the SPLS variants (as described in Definition 9). Note additionally that the subsignature relation for SPLSs, like the one for program signatures, is reflexive, transitive and not anti-symmetric.

Example 5 (CapAccInt and FinAccInt SPLSs). Figure 6a gives an interface of the CapitalAccount SPL (see Figure 2), named CapAccInt, constructed by hiding the features BalanceInfo and YearlyFees from CapitalAccount. Additionally, Figure 6b gives an interface of the FinancialAccount SPL (see Figure 4), named FinAccInt, constructed by hiding the feature AmountInfo from FinancialAccount.

3.3. Dependent SPLs and Multi SPLs

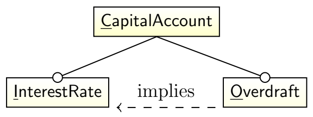
A *Dependent SPL* (DPL) is an SPL extended with dependencies modeled by SPLSs. The abstract syntax of DPLs is given in Figure 7. A DPL declaration comprises the name $\mathtt{L}$ of the DPL, a sequence of SPLS names $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ specifying its dependencies, a pair of feature models $\mathcal{M}^{Main}$ and $\mathcal{M}^{Glue}$, a configuration knowledge $\mathcal{K}$ and an artifact base AB. The actual feature model of $\mathtt{L}$, denoted by $\mathcal{M}_{\mathtt{L}}$, is structured in 2+n parts: the feature model $\mathcal{M}^{Main}$, which describes the part of $\mathcal{M}_{\mathtt{L}}$ that is local to $\mathtt{L}$; the feature model $\mathcal{M}^{Glue}$, which describes how the local features of $\mathtt{L}$ are related with the features of $\mathtt{L}$'s dependencies; and the feature models of $\mathtt{L}$'s dependencies $\mathcal{M}_{\mathtt{Z}_1}, \ldots, \mathcal{M}_{\mathtt{Z}_n}$. Namely, the feature model of $\mathtt{L}$ is defined as a composition of $\mathcal{M}^{Main}$ and the feature models $\mathcal{M}_{\mathtt{Z}_1}, \ldots, \mathcal{M}_{\mathtt{Z}_n}$, glued together with $\mathcal{M}^{Glue}$: $\mathcal{M}_{\mathtt{L}} = \mathcal{M}^{Main/\overline{\mathtt{Z}}} = \mathcal{M}^{Main/\mathtt{Z}_1,\ldots\mathtt{Z}_n} = \mathcal{M}^{Main} \circ_{\mathcal{M}^{Glue}} (\mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n}))$.

The n SPLSs $\overline{\mathtt{Z}}$ are pairwise different and each class can only be declared and modified by at most one of $AB, AB_{\mathtt{Z}_1^\star}, \ldots AB_{\mathtt{Z}_n^\star}$. This condition rules out class name clashes between the artifact base of $\mathtt{L}$ and its dependencies.

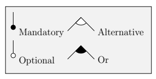
The *nucleus of the DPL* $\mathtt{L}$ is the SPL, denoted as $\mathcal{N}(\mathtt{L})$, that describes the classes and attributes declared locally to $\mathtt{L}$. Namely, $\mathcal{N}(\mathtt{L})$ is the SPL with feature model $\mathcal{M}_{\mathtt{L}}$, configuration knowledge $\mathcal{K}$, and artifact base AB.

With this last piece added to our model, we can finally define our notion of Multi Software Product Line.

Definition 11 (Multi SPL). A *Multi Software Product Line* is a collection of SPLs, SPL signatures, and Dependent SPLs.



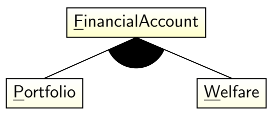
$$\mathcal{F}_{\mathsf{CapAccInt}} = \{\ \mathsf{C}\ \ \mathsf{I}\ \ \mathsf{O}\ \}$$
$$\mathcal{P}_{\mathsf{CapAccInt}} = \{\{\ \mathsf{C}\ \ \mathsf{I}\ \}, \{\ \mathsf{C}\ \}, \{\ \mathsf{C}\ \ \mathsf{I}\ \ \mathsf{O}\ \}\}$$



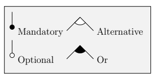
$<_{\mathsf{CapAccInt}}$: {`dSigInterest`, `dSigOverdraft`}
$\alpha_{\mathsf{CapAccInt}}$: `dSigInterest` $\mapsto$ I, `dSigOverdraft` $\mapsto$ O

```
class CapAccount extends Object {                  // Base Program
          String identity;
          double balance;
          Date lastUpdate;
          void withdraw(double x);
}
delta dSigInterest{                                // Deltas
        modifies class CapAccount {
                  adds double yearRate;
                  adds double opFees;
                  adds void interestUpdate(double rate);
        }
}
delta dSigOverdraft{
        modifies class CapAccount {
                  adds double maxOver;
                  adds double negativeRate;
                  adds void negUpdate();
        }
}
```

(a) CapAccInt SPLS: $\mathcal{M}_{\mathsf{CapAccInt}}$ (left), $\mathcal{K}_{\mathsf{CapAccInt}}$ (right top), and $ABS_{\mathsf{CapAccInt}}$ (right bottom).



$$\mathcal{F}_{\mathsf{FinAccInt}} = \{\ \mathsf{F}\ \ \mathsf{P}\ \ \mathsf{W}\ \}$$
$$\mathcal{P}_{\mathsf{FinAccInt}} = \{\{\ \mathsf{F}\ \ \mathsf{P}\ \}, \{\ \mathsf{F}\ \ \mathsf{W}\ \}, \{\ \mathsf{F}\ \ \mathsf{P}\ \ \mathsf{W}\ \}\}$$



$<_{\mathsf{FinAccInt}}$: {`dSigPortfolio`, `dSigWelfare`}
$\alpha_{\mathsf{FinAccInt}}$: `dSigPortfolio` $\mapsto$ P, `dSigWelfare` $\mapsto$ W

```
class FinAccount extends Object {                 // Base program
         String identity; double liquidity;
}
delta dSigPortfolio{                              // Deltas
        adds class RiskProd extends Object {
                      String info;
                      int quantity; }
        modifies class FinAccount {
                 adds LinkedList portfolio;
                 adds void addToPortfolio(String i, int q);
        }
}
delta dSigWelfare {
        adds class LifeProd extends Object {
                      String info;
                      String beneficiary; }
    modifies class FinAccount {
             adds ArrayList welfare;
             adds void addToWelfare(String i, String b);
    }
}
```

(b) FinAccInt SPLS: $\mathcal{M}_{\mathsf{FinAccInt}}$ (left), $\mathcal{K}_{\mathsf{FinAccInt}}$ (right top), and $ABS_{\mathsf{FinAccInt}}$ (right bottom).

Figure 6: The SPLSs CapAccInt and FinAccInt

LD ::= **line** L $(\overline{\mathtt{Z}})$ $\{\mathcal{M}^{Main}$ $\mathcal{M}^{Glue}$ $\mathcal{K}$ $AB\}$ Dependent SPL Declaration

Figure 7: Dependent SPLs. The extensions with respect to IFΔJ SPLs (given in Figure 1b, with the syntax of artifact bases AB) are highlighted in grey.

Example 6 (An MPL of accounts). Consider the MPL that includes the product lines CapitalAccount and FinancialAccount, and the SPL signatures CapAccInt and FinAccInt. We can add to it the Dependent SPL DualAccount, defined as follows, that can mix any implementations of CapAccInt and FinAccInt together:

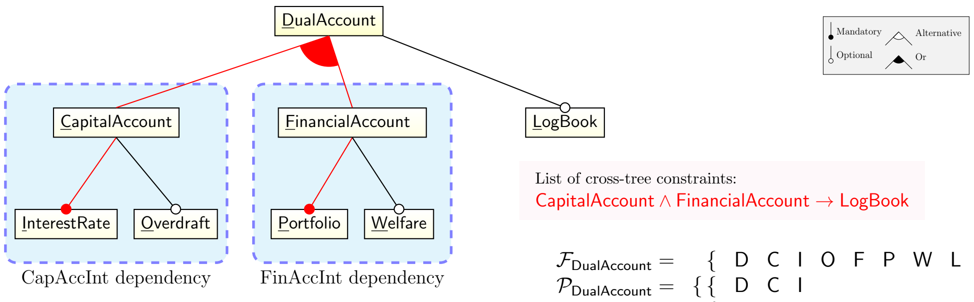
line DualAccount(CapAccInt,FinAccInt) $\{\mathcal{M}^{Main}_{\mathsf{DualAccount}}$ $\mathcal{M}^{Glue}_{\mathsf{DualAccount}}$ $\mathcal{K}_{\mathsf{DualAccount}}$ $AB_{\mathsf{DualAccount}}\}$

The feature models $\mathcal{M}^{Main}_{\mathsf{DualAccount}}$ and $\mathcal{M}^{Glue}_{\mathsf{DualAccount}}$ are described in Figure 8 (together with the feature model $\mathcal{M}_{\mathsf{DualAccount}}$), while the configuration knowledge $\mathcal{K}_{\mathsf{DualAccount}}$ and the artifact base $AB_{\mathsf{DualAccount}}$ are described in Figure 9. This DPL provides a class `DualAccount` that combines two bank accounts satisfying the dependencies CapAccInt (given in Figure 6a) and FinAccInt (given in Figure 6b), respectively. Its feature model introduces an optional feature LogBook that logs the activities of an account. It also relates this feature to the feature models of CapAccInt and FinAccInt with the $\mathcal{M}^{Glue}_{\mathsf{DualAccount}}$ feature model that expresses the following constraint:

$$\begin{gathered}\mathsf{FinancialAccount} \vee \mathsf{CapitalAccount}\\ \mathsf{CapitalAccount} \rightarrow \mathsf{InterestRate}\\ \mathsf{FinancialAccount} \rightarrow \mathsf{Portfolio}\\ \mathsf{CapitalAccount} \wedge \mathsf{FinancialAccount} \rightarrow \mathsf{LogBook}\end{gathered}$$

This constraint means that at least one backend for the `DualAccount` class must be selected, all capital accounts will have the `InterestRate` feature, all financial accounts will have the `Portfolio` feature, and if both capital and financial account functionalities are provided by the dual account, then the LogBook feature must be selected to register all possible activity on the account.

Remark 12 (Order in dependency list). It is interesting to note that (since the join operation of feature models is associative and commutative) the order in which the SPLS names $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ are listed in a DPL declaration (see Figure 7) is immaterial.



$$\begin{aligned}
\mathcal{F}^{Main}_{\mathsf{DualAccount}} &= \{\ \mathsf{D}\ \mathsf{L}\ \} \\
\mathcal{P}^{Main}_{\mathsf{DualAccount}} &= \{\{\ \mathsf{D}\ \},\ \{\ \mathsf{D}\ \mathsf{L}\ \}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}^{Glue}_{\mathsf{DualAccount}} &= \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \} \\
\mathcal{P}^{Glue}_{\mathsf{DualAccount}} &= \{\{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \},\ \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{\mathsf{DualAccount}} &= \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{O}\ \mathsf{F}\ \mathsf{P}\ \mathsf{W}\ \mathsf{L}\ \} \\
\mathcal{P}_{\mathsf{DualAccount}} &= \{\{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{O}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{O}\ \mathsf{L}\ \}, \\
&\quad \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \},\ \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \mathsf{W}\ \},\ \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{F}\ \mathsf{P}\ \mathsf{W}\ \mathsf{L}\ \}, \\
&\quad \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{O}\ \mathsf{F}\ \mathsf{P}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{F}\ \mathsf{P}\ \mathsf{W}\ \mathsf{L}\ \},\ \{\ \mathsf{D}\ \mathsf{C}\ \mathsf{I}\ \mathsf{O}\ \mathsf{F}\ \mathsf{P}\ \mathsf{W}\ \mathsf{L}\ \}\}
\end{aligned}$$

Figure 8: DualAccount DPL: feature model $\mathcal{M}_{\mathsf{DualAccount}}$.

Remark 13 (DPL conservatively extends SPL). We can ensure that the concept of DPL is a conservative extension of the concept of SPL (cf. Section 2.1), by assuming that if a DPL $\mathtt{L}$ has no dependencies (i.e., $\overline{\mathtt{Z}} = \emptyset$) then $\mathcal{M}^{Glue} = \mathcal{M}_{Id}$. Therefore (since $\mathcal{M}_{Id}$ is the neutral element of the join operation of feature models): (i) any DPL $\mathtt{L}$ without dependencies can be seen as an SPL with feature model $\mathcal{M}_{\mathtt{L}} = \mathcal{M}^{Main}$; and (ii) any SPL $\mathtt{L}$ can be seen as a DPL with $\mathcal{M}^{Main} = \mathcal{M}_{\mathtt{L}}$ and $\mathcal{M}^{Glue} = \mathcal{M}_{Id}$.

Remark 14 (On the scope of delta names). To simplify the manipulation of our model in the rest of the document, without loss of generality, we assume that the scope of the name of a delta is limited to the DPL or SPLS that contain its declaration—i.e., delta names are bound by DPL and SPLS declarations. In particular, in the following, when composing different SPLs together, we consider that α-conversion is implicitly applied and all deltas have different names.

4. Composing SPLs

In the previous section we presented our model of MPL with its different components, SPL, SPL signatures and Dependent SPL. We now describe

$<_{\mathsf{DualAccount}}$: $\{\mathtt{dDualC}, \mathtt{dDualF}, \mathtt{dDualP}, \mathtt{dDualW}, \mathtt{dLog}\} < \{\mathtt{dLogC}, \mathtt{dLogP}, \mathtt{dLogW}\}$
$\alpha_{\mathsf{DualAccount}}$: $\mathtt{dDualC} \mapsto \mathsf{C}, \mathtt{dDualF} \mapsto \mathsf{F}, \mathtt{dDualP} \mapsto \mathsf{P}, \mathtt{dDualW} \mapsto \mathsf{W},$
$\mathtt{dLog} \mapsto \mathsf{L}, \mathtt{dLogC} \mapsto (\mathsf{C} \wedge \mathsf{L}), \mathtt{dLogP} \mapsto (\mathsf{P} \wedge \mathsf{L}), \mathtt{dLogW} \mapsto (\mathsf{W} \wedge \mathsf{L})$

```
class DualAccount extends Object { String identity; void setId(String id){identity=id}; } // Base program

delta dDualC {                                                                            // Deltas
        modifies class DualAccount extends Object {
                adds CapAccount cap=new CapAccount();
                adds void withdraw(double x){ cap.withdraw(x); }
                modifies void setId(String id){ cap.identity=id; original(id); }
        }
}
delta dDualF {
        modifies class DualAccount extends Object {
                adds FinAccount fin=new FinAccount();
                modifies void setId(String id){ fin.identity=id; original(id); }
        }
}
delta dDualP {
        modifies class DualAccount extends Object {
                adds void add2P(String i, int q){ fin.portfolio.addToPortfolio(i,q); }
        }
}
delta dDualW {
        modifies class DualAccount extends Object {
                adds void add2W(String i, String b){ fin.welfare.addToWelfare(i,b); }
        }
}
delta dLog {
        modifies class DualAccount extends Object {
                adds String journalLog;
        }
}
delta dLogC {
        modifies class DualAccount extends Object {
                modifies void withdraw(double x){ journalLog+= "::withdraw("+x+")"; original(x); }
        }
}
delta dLogP {
        modifies class DualAccount extends Object {
                modifies void add2P(String i, int q){
                                journalLog+= "::add2P("+i+","+q+")";
                                original(i, q); }
        }
}
delta dLogW {
        modifies class DualAccount extends Object {
                modifies void add2W(String i, String b){
                                journalLog+= "::add2W("+i+","+b+")";
                                original(i, b); }
        }
}
```

Figure 9: DualAccount DPL: configuration knowledge $\mathcal{K}_{\mathsf{DualAccount}}$ (top); and artifact base $AB_{\mathsf{DualAccount}}$ (bottom).

two composition mechanisms to build complex product lines from these components. The first composition mechanism, called *DPL-SPLs composition*, constructs SPLs by filling the dependencies of a Dependent SPL. The second mechanism extends the first one, allowing to combine Dependent SPL together.

Additionally, these two composition mechanisms are modular with respect to several formal analysis on SPLs. In particular, we show in Section 4.3 that our mechanisms allows several analysis of feature models (void feature model, core features, dead features, void partial configuration, and atomic sets) to be checked modularly; and we show in Section 4.4 that type checking can also be performed modularly by using existing type checking tools.

4.1. DPL-SPLs composition

This first composition mechanism allows to construct a new SPL by taking a Dependent SPL and filling its dependencies, modeled with SPL signatures, with SPLs that implement such signatures. This mechanism extends the notion of feature model composition (see Definition 6) to encompass the configuration knowledge and the artifact base, and like the feature model composition, it is structured in three operators: $\mathcal{R}$, $\bullet$ and the actual composition operator (noted $\circ$ for feature models). We present these operators for the DPL-SPLs composition in the following definitions.

Definition 15 (Operation $\mathcal{R}$ on SPLs). Let $\mathtt{L}$ be an SPL. The SPL $\mathcal{R}(\mathtt{L})$ is $\mathtt{L}$ if $\emptyset \in \mathcal{P}_{\mathtt{L}}$, otherwise is defined as follow:

1. feature model $\mathcal{M}_{\mathcal{R}(\mathtt{L})} = (\mathcal{F}_{\mathtt{L}}, \mathcal{P}_{\mathtt{L}} \cup \{\emptyset\})$;
2. artifact base $AB_{\mathcal{R}(\mathtt{L})}$ obtained from $AB_{\mathtt{L}}$ by transforming the base program into a delta (the *base* delta) with fresh name $\mathtt{d}_{\mathtt{L}}$ and by adding an empty base program; and
3. configuration knowledge $\mathcal{K}_{\mathcal{R}(\mathtt{L})}$ obtained from $\mathcal{K}_{\mathtt{L}}$ by defining: (i) $\alpha_{\mathcal{R}(\mathtt{L})}$ as the minimal extension of $\alpha_{\mathtt{L}}$ such that $\alpha_{\mathcal{R}(\mathtt{L})}(\mathtt{d}_{\mathtt{L}}) = \mathcal{P}_{\mathtt{L}}$, and (ii) $<_{\mathcal{R}(\mathtt{L})}$ as the minimal extension of $<_{\mathtt{L}}$ that has $\mathtt{d}_{\mathtt{L}}$ as bottom.

Similarly to its definition for feature models, the $\mathcal{R}$ operator takes an SPL $\mathtt{L}$ in input, returns it unchanged if the empty product $\emptyset$ is a valid product for $\mathtt{L}$, or extends it with the empty product $\emptyset$ which is mapped to the empty variant by the generator $\mathcal{G}_{\mathcal{R}(\mathtt{L})}$.

Definition 16 (Operation $\bullet$ on SPLs). Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two SPLs such that each class can only be declared and modified by at most one of $AB_{\mathtt{L}_1}$ and $AB_{\mathtt{L}_2}$.[3] The join of $\mathtt{L}_1$ and $\mathtt{L}_2$, denoted as $\mathtt{L}_1\bullet\mathtt{L}_2$, is the SPL $\mathtt{L}$ such that:

1. $\mathcal{M}_{\mathtt{L}} = \mathcal{M}_{\mathtt{L}_1}\bullet\mathcal{M}_{\mathtt{L}_2}$;
2. $\mathcal{K}_{\mathtt{L}} = ((\alpha'_{\mathtt{L}_1} \cup \alpha'_{\mathtt{L}_2}), (<_{\mathtt{L}_1} \cup <_{\mathtt{L}_2}))$ where
   - $\alpha'_{\mathtt{L}_i}(\mathtt{d}) = \{p \in \mathcal{P}_{\mathtt{L}} \mid p \cap \mathcal{F}_{\mathtt{L}_i} \in \alpha_{\mathtt{L}_i}(\mathtt{d})\}$ for all deltas $\mathtt{d}$ of $\mathtt{L}_i$;
3. $AB_{\mathtt{L}} = AB_{\mathtt{L}_1} \cup AB_{\mathtt{L}_2}$.

Similarly to its definition for feature models, the $\bullet$ operator simply combines the two SPLs in input. Note that the $\bullet$ operator on SPLs is associative and commutative (like the corresponding operator on feature models), with the following SPL $\mathtt{L}_\emptyset$ as neutral element: **line** $\mathtt{L}_\emptyset$ $\{$ $\mathcal{M}_{Id}$ $(\emptyset, \emptyset)$ $\emptyset$ $\}$.

Definition 17 (DPL-SPLs composition). Let $\mathtt{L}$ be a DPL with dependencies $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ $(n \geq 0)$. Let $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ be SPLs such that:

(i) $\mathtt{Z}_i \preceq \mathtt{L}_i$ $(1 \leq i \leq n)$, and
(ii) the n SPLs $\overline{\mathtt{L}}$ are such that each class can only be declared and modified by at most one of $AB_{\mathtt{L}}, AB_{\mathtt{L}_1}, ...AB_{\mathtt{L}_n}$.

The composition of $\mathtt{L}$ with $\overline{\mathtt{L}}$, is the SPL $\mathtt{L}(\overline{\mathtt{L}}) = \mathcal{N}(\mathtt{L})\bullet\mathcal{R}(\mathtt{L}_1)\bullet\ldots\bullet\mathcal{R}(\mathtt{L}_n)$.

Note that, if $\mathtt{L}$ has no dependencies (i.e., $n = 0$), then $\mathcal{G}_{\mathtt{L}(\overline{\mathtt{L}})} = \mathcal{G}_{\mathtt{L}(\emptyset)} = \mathcal{G}_{\mathtt{L}}$ (so, $\mathtt{L}(\overline{\mathtt{L}})$ and $\mathtt{L}$ have the same variants).

Example 7 (DualAccount composition). The DPL DualAccount (see Example 6) can be composed with the SPLs CapitalAccount and FinancialAccount to obtain the SPL DualAccount(CapitalAccount,FinancialAccount). Indeed, by construction, the SPLs CapitalAccount and FinancialAccount do not declare or modify a same class. Moreover, as shown in the Example 5, the SPL CapitalAccount implements the SPL signature CapAccInt, and the SPL FinancialAccount implements the SPL signature FinAccInt.

We can thus apply the DPL-SPLs composition to obtain an SPL CompleteAccount that mixes the functionalities declared in CapitalAccount and FinancialAccount in the way specified by the DPL DualAccount. The feature

[3]This assumption enforces a boundary between the different SPLs involved in the composition and rules out class name clashes between variants of different SPLs—cf. the assumption in the second paragraph of Section 3.3.

model of the CompleteAccount SPL is depicted in Figure 10. Intuitively, this feature model is obtained by replacing the CapAccInt and FinAccInt sub-feature models by the ones from CapitalAccount and FinancialAccount.

Note however that the constraints added by DualAccount has several effects on the feature model of CapitalAccount (written in gray in Figure 10): the feature `YearlyFees` cannot be selected anymore (i.e., it is a *dead feature*), because it conflicts with `InterestRate` which became mandatory; and the implication between the `Overdraft` feature and `InterestRate` is lost, again because `InterestRate` is now mandatory. A similar effect occurs on the `Welfare` feature which is now simply optional, as `Porfolio` is now mandatory.

The configuration knowledge and the artifact base of the CompeteAccount SPL are a simple union of the ones of DualAccount, CapitalAccount and FinancialAccount, with the only difference that the base programs of CapitalAccount and FinancialAccount have been replaced by new deltas.

4.1.1. Feature Model of the DPL-SPLs composition

As discussed in the Example 7, the construction of the feature model of the CompleteAccount SPL involves the composition of six feature models:

$$\begin{aligned}\mathcal{M}_{\mathsf{CompleteAccount}} = \mathcal{M}^{Main}_{\mathsf{DualAccount}} \bullet \mathcal{M}^{Glue}_{\mathsf{DualAccount}} \bullet \mathcal{R}(\mathcal{M}_{\mathsf{CapAccInt}}) \\ \bullet \mathcal{R}(\mathcal{M}_{\mathsf{FinAccInt}}) \bullet \mathcal{R}(\mathcal{M}_{\mathsf{CapitalAccount}}) \bullet \mathcal{R}(\mathcal{M}_{\mathsf{FinancialAccount}}) \quad (1)\end{aligned}$$

In general, in order to compute the feature model of a DPL-SPLs composition, the number of feature models to compose is equal to $2 + 2n$ where n is the number of the DPL's dependencies: 2 for the $\mathcal{M}^{Main}$ and $\mathcal{M}^{Glue}$, n for the dependencies, and n for the implementation of these dependencies. This number can be however reduced to $2 + n$ using the following lemma.

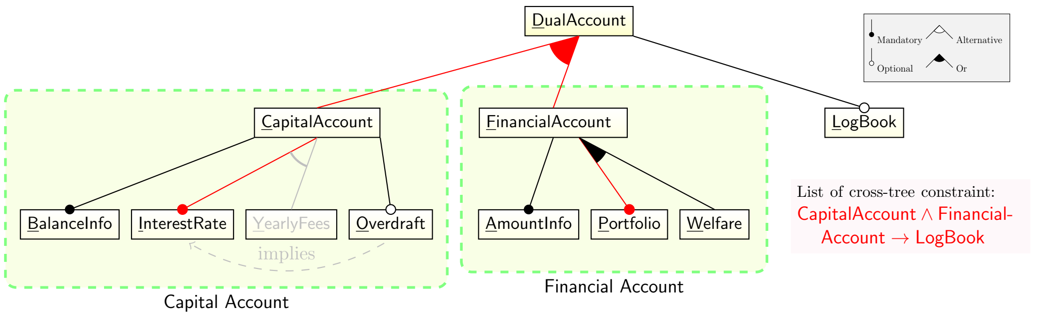
Lemma 1 (Feature model interface relation and operations $\mathcal{R}$ and $\bullet$). *Consider two feature models $\mathcal{M}_x$ and $\mathcal{M}_y$ such that $\mathcal{M}_y \preceq \mathcal{M}_x$. Then we have $\mathcal{R}(\mathcal{M}_y) \preceq \mathcal{R}(\mathcal{M}_x)$ and $\mathcal{M}_y \bullet \mathcal{M}_x = \mathcal{M}_x$*

Proof. See Appendix B. □

Using this lemma on the Equation 1, we obtain that:

$$\begin{cases} \mathcal{R}(\mathcal{M}_{\mathsf{CapAccInt}}) \bullet \mathcal{R}(\mathcal{M}_{\mathsf{CapitalAccount}}) = \mathcal{R}(\mathcal{M}_{\mathsf{CapitalAccount}}) \\ \mathcal{R}(\mathcal{M}_{\mathsf{FinAccInt}}) \bullet \mathcal{R}(\mathcal{M}_{\mathsf{FinancialAccount}}) = \mathcal{R}(\mathcal{M}_{\mathsf{FinancialAccount}}) \end{cases}$$

and so $\mathcal{M}_{\mathsf{CompleteAccount}} = \mathcal{M}^{Main}_{\mathsf{DualAccount}} \bullet \mathcal{M}^{Glue}_{\mathsf{DualAccount}} \bullet \mathcal{R}(\mathcal{M}_{\mathsf{CapitalAccount}}) \bullet \mathcal{R}(\mathcal{M}_{\mathsf{FinancialAccount}})$



$$
\begin{array}{ll}
\mathcal{F}_{\mathsf{DualAccount}} = & \{\mathsf{D\ C\ I\ O\ F\ P\ W\ L}\} \\
\mathcal{P}_{\mathsf{DualAccount}} = & \{\{\mathsf{D\ C\ I}\}, \\
& \{\mathsf{D\ C\ I\ O}\}, \\
& \{\mathsf{D\ C\ I\ L}\}, \\
& \{\mathsf{D\ C\ I\ O\ L}\}, \\
& \{\mathsf{D\ F\ P}\}, \\
& \{\mathsf{D\ F\ P\ W}\}, \\
& \{\mathsf{D\ F\ P\ L}\}, \\
& \{\mathsf{D\ F\ P\ W\ L}\}, \\
& \{\mathsf{D\ C\ I\ F\ P\ L}\}, \\
& \{\mathsf{D\ C\ I\ O\ F\ P\ L}\}, \\
& \{\mathsf{D\ C\ I\ F\ P\ W\ L}\}, \\
& \{\mathsf{D\ C\ I\ O\ F\ P\ W\ L}\}\}
\end{array}
\qquad
\begin{array}{ll}
\mathcal{F}_{\mathsf{CompleteAccount}} = & \{\mathsf{D\ C\ B\ I\ Y\ O\ F\ A\ P\ W\ L}\} \\
\mathcal{P}_{\mathsf{CompleteAccount}} = & \{\{\mathsf{D\ C\ B\ I}\}, \\
& \{\mathsf{D\ C\ B\ I\ O}\}, \\
& \{\mathsf{D\ C\ B\ I\ L}\}, \\
& \{\mathsf{D\ C\ B\ I\ O\ L}\}, \\
& \{\mathsf{D\ F\ A\ P}\}, \\
& \{\mathsf{D\ F\ A\ P\ W}\}, \\
& \{\mathsf{D\ F\ A\ P\ L}\}, \\
& \{\mathsf{D\ F\ A\ P\ W\ L}\}, \\
& \{\mathsf{D\ C\ B\ I\ F\ A\ P\ L}\}, \\
& \{\mathsf{D\ C\ B\ I\ O\ F\ A\ P\ L}\}, \\
& \{\mathsf{D\ C\ B\ I\ F\ A\ P\ W\ L}\}, \\
& \{\mathsf{D\ C\ B\ I\ O\ F\ A\ P\ W\ L}\}\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{F}_{\mathcal{R}(\mathsf{CapitalAccount})} = & \{\mathsf{C\ B\ I\ Y\ O}\} \\
\mathcal{P}_{\mathcal{R}(\mathsf{CapitalAccount})} = & \{\{\ \}, \\
& \{\mathsf{C\ B\ I}\}, \\
& \{\mathsf{C\ B\ Y}\}, \\
& \{\mathsf{C\ B\ I\ O}\}\}
\end{array}
\qquad
\begin{array}{ll}
\mathcal{F}_{\mathcal{R}(\mathsf{FinancialAccount})} = & \{\mathsf{F\ A\ P\ W}\} \\
\mathcal{P}_{\mathcal{R}(\mathsf{FinancialAccount})} = & \{\{\ \}, \\
& \{\mathsf{F\ A\ P}\}, \\
& \{\mathsf{F\ A\ W}\}, \\
& \{\mathsf{F\ A\ P\ W}\}\}
\end{array}
$$

Figure 10: CompleteAccount DPL: feature model $\mathcal{M}_{\mathsf{CompleteAccount}}$.

Hence in general, in order to compute the feature model of a DPL-SPLs composition, it is necessary to compose only $2+n$ feature models, where n is the number of the DPL's dependencies: 2 for the $\mathcal{M}^{Main}$ and $\mathcal{M}^{Glue}$, and n for the implementation of the DPL's dependencies. This property is formally stated in the following theorem.

Theorem 2 (Feature Model of the DPL-SPLs composition). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ *($n \geq 0$), and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ *be SPLs such that the composition* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ *is defined. Then we have*

$$\mathcal{M}_{\mathtt{L}_0} = \mathcal{M}^{Main}_{\mathtt{L}} \bullet \mathcal{M}^{Glue}_{\mathtt{L}} \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \ldots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n}))$$

Proof. See Appendix B. □

4.1.2. Generator of the DPL-SPLs composition

As discussed in the Example 7, the configuration knowledge and the artifact base of the SPL resulting from a DPL-SPLs composition $\mathtt{L}(\overline{\mathtt{L}})$ is the union (sligthly modified) of all the SPLs and DPL that took part in the composition. It can thus be difficult to imagine what could be the variant associated to a product.

The following theorem shed light on the nature of the generator of an SPL resulting from a DPL-SPLs composition $\mathtt{L}(\overline{\mathtt{L}})$: it states that the variants of such composition can be generated by directly using the generator of the DPL $\mathtt{L}$, the generators of the SPLs $\overline{\mathtt{L}}$, and merging the generated variants together. An interesting result from this property is that there is no need to actually build the whole $\mathtt{L}(\overline{\mathtt{L}})$ SPL when performing a DPL-SPLs composition, computing the feature model $\mathcal{M}_{\mathtt{L}(\overline{\mathtt{L}})}$ is enough.

Theorem 3 (Generator of the DPL-SPLs composition). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ *(*$n \geq 0$*), and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ *be SPLs such that the composition* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ *is defined and, for all* $\mathtt{L}_i \in \overline{\mathtt{L}}$ *and all* $p \in \mathrm{dom}(\mathcal{G}_{\mathtt{L}_i}) \cup \{\emptyset\}$*, let*

$$\mathcal{G}^{\emptyset}_{\mathtt{L}_i}(p) = \begin{cases} \emptyset & \text{if } p = \emptyset \text{ and } p \notin \mathcal{P}_{\mathtt{L}_i} \\ \mathcal{G}_{\mathtt{L}_i}(p) & \text{otherwise.} \end{cases}$$

For every product $p \in \mathrm{dom}(\mathcal{G}_{\mathtt{L}_0})$*, we have*

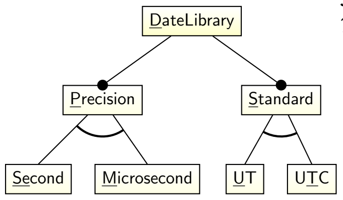
$$\mathcal{G}_{\mathtt{L}_0}(p) = \mathcal{G}_{\mathcal{N}(\mathtt{L})}(p \cap \mathcal{F}_{\mathtt{L}}) \cup \big(\bigcup_{\mathtt{L}_i \in \overline{\mathtt{L}}} \mathcal{G}^{\emptyset}_{\mathtt{L}_i}(p \cap \mathcal{F}_{\mathtt{L}_i}) \big)$$

Proof. See Appendix C. □

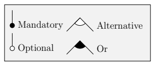
4.1.3. Modularity in the DPL-SPLs composition

As the DPL-SPLs composition takes SPL in arguments, and generates a new SPL, this new SPL can itself be used as an argument of another DPL-SPLs composition. Consider for instance the following DPL example.

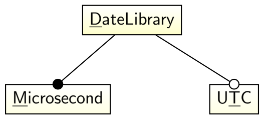
Example 8 (The DepCapitalAccount DPL). Suppose that after the creation of the CapitalAccount SPL, someone developed an SPL of date, called DateLibrary, where one can choose the time precision of the provided PrecisionDate class. As time precision can be very important for account management, it makes sense to refactor the CapitalAccount SPL into a new DPL, called DepCapitalAccount, that depends on functionalities provided by this library or others like it. We moreover suppose that the refactoring is done in a way such that the signature of $\mathcal{N}(\mathsf{DepCapitalAccount})$ is equalt to the signature of CapitalAccount. We illustrate the feature models of the DateLibrary SPL, of the DepCapitalAccount DPL and its dependency in Figures 11a, 11c and 11b respectively.



$$\begin{aligned}\mathcal{F}_{\mathsf{DateLibrary}} &= \{\ \mathsf{D}\ \mathsf{P}\ \mathsf{S}\ \mathsf{Se}\ \mathsf{M}\ \mathsf{U}\ \mathsf{T}\ \}\\ \mathcal{P}_{\mathsf{DateLibrary}} &= \{\{\ \mathsf{D}\ \mathsf{P}\ \mathsf{S}\ \mathsf{Se}\ \mathsf{U}\ \},\\ &\quad\ \{\ \mathsf{D}\ \mathsf{P}\ \mathsf{S}\ \mathsf{M}\ \mathsf{U}\ \},\\ &\quad\ \{\ \mathsf{D}\ \mathsf{P}\ \mathsf{S}\ \mathsf{Se}\ \mathsf{T}\ \},\\ &\quad\ \{\ \mathsf{D}\ \mathsf{P}\ \mathsf{S}\ \mathsf{M}\ \mathsf{T}\ \}\ \}\end{aligned}$$

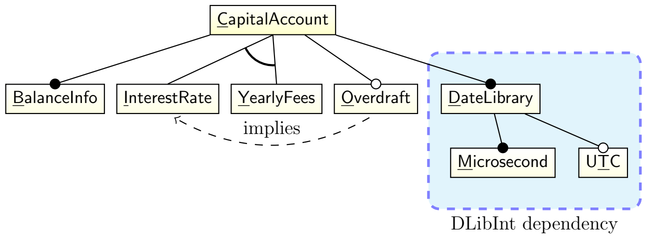


(a) DateLibrary SPL: Feature Model



$$\begin{aligned}\mathcal{F}_{\mathsf{DLibInt}} &= \{\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \}\\ \mathcal{P}_{\mathsf{DLibInt}} &= \{\{\ \mathsf{D}\ \mathsf{M}\ \},\\ &\quad\ \{\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \}\ \}\end{aligned}$$

(b) The dependency of DepCapitalAccount, called DLibInt: Feature Model



$$\begin{aligned}\mathcal{F}^{Main}_{\mathsf{DepCapitalAccount}} &= \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{Y}\ \mathsf{O}\ \}\\ \mathcal{P}^{Main}_{\mathsf{DepCapitalAccount}} &= \{\{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{Y}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{O}\ \}\ \}\end{aligned}$$

$$\begin{aligned}\mathcal{F}^{Glue}_{\mathsf{DepCapitalAccount}} &= \{\ \mathsf{C}\ \mathsf{D}\ \}\\ \mathcal{P}^{Glue}_{\mathsf{DepCapitalAccount}} &= \{\{\ \mathsf{C}\ \mathsf{D}\ \}\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_{\mathsf{DepCapitalAccount}} &= \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{Y}\ \mathsf{O}\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \}\\ \mathcal{P}_{\mathsf{DepCapitalAccount}} &= \{\{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{D}\ \mathsf{M}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{Y}\ \mathsf{D}\ \mathsf{M}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{O}\ \mathsf{D}\ \mathsf{M}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{Y}\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \},\\ &\quad\ \{\ \mathsf{C}\ \mathsf{B}\ \mathsf{I}\ \mathsf{O}\ \mathsf{D}\ \mathsf{M}\ \mathsf{T}\ \}\ \}\end{aligned}$$

(c) The DepCapitalAccount SPL: Feature Model

Figure 11: Feature Models of DateLibrary and DepCapitalAccount

Using the DPL-SPL composition mechanism, we can compose this new DepCapitalAccount DPL, which is a more flexible version of the CapitalAccount, with the DateLibrary SPL to obtain an SPL equivalent to CapitalAccount. Intuitively, it makes sense that this new SPL implements the CapAccInt signature and so could be used as parameter to the DualAccount DPL. The following definition and theorem formalize this intuition.

Definition 18 (Interface relation for DPLs). An SPLS $\mathtt{Z}_{\mathit{Int}}$ is an *interface* of a DPL $\mathtt{L}$ (or, equivalently, DPL $\mathtt{L}$ implements SPLS $\mathtt{Z}_{\mathit{Int}}$), denoted $\mathtt{Z}_{\mathit{Int}} \preceq \mathtt{L}$, iff $\mathtt{Z}_{\mathit{Int}}$ is an interface of $\mathcal{N}(\mathtt{L})$.

Note that in the Example 8, the signature of the SPL $\mathcal{N}$(DepCapitalAccount) is equal to the signature of CapitalAccount. Therefore DepCapitalAccount implements CapAccInt.

The following theorem states that fulfilling the dependencies of a DPL preserves the set of implemented interfaces.

Theorem 4 (Interfaces of DPL-SPLs composition). *Let* $\mathtt{L}$ *be a DPL implementing an SPLS* $\mathtt{Z}$*, with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ *(*$n \geq 0$*). Let* $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_n$ *be SPLs such that the composition* $\mathtt{L}(\overline{\mathtt{L}})$ *is defined. We then have that* $\mathtt{Z} \preceq \mathtt{L}(\overline{\mathtt{L}})$*.*

Proof. See Appendix D. □

Following our previous analysis that the DPL DepCapitalAccount implements the signature CapAccInt, the Theorem 4 tells us that the SPL DepCapitalAccount(DateLibrary) also implements the signature CapAccInt. Consequently, the following DPL-SPLs composition is defined:

$$\text{DualAccount}(\,\text{DepCapitalAccount}(\text{DateLibrary}),\ \text{FinancialAccount})\quad .$$

4.2. DPL-DPLs composition

The notion of DPL-DPLs composition extends the notion of DPL-SPLs composition (Definition 17) to accept DPLs as arguments and yielding a DPL as result. Such a composition mechanism is interesting in two main cases. First, it allows to combine different DPLs, i.e., description on how to merge different SPLs, without forcing to set these SPLs. Consider for instance the DPL-SPLs composition presented in Section 4.1.3: it could be interesting to construct a DPL of a CompleteAccount where the implementation of `Date` is still free. Second, this mechanism is also interesting as it allows for using one unique SPL to satisfy the dependencies of several DPL. Consider again the the DPL-SPLs composition presented in Section 4.1.3, and in particular the DPL DualAccount (presented in the Example 6): this DPL implements a Log mechanism that records the activities on its account, without recording at what time these activities occur. It would thus make perfect sense to refactor DualAccount into a new DPL DepDualAccount that also depends on the DLibInt signature, and uses its functionalities to record the date of each activity. That way, it would be possible to construct a new version of a CompleteAccount SPL using the DepDualAccount and the DepCapitalAccount DPLs, with the DateLibrary SPL shared between the two DPLs.

Definition 19 (DPL-DPLs composition). Let $\mathtt{L}$ be a DPL with dependencies $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ $(n \geq 0)$. Let $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_m$ $(m \leq n)$ be DPLs with respective dependencies $\overline{\mathtt{Z}}^{(i)} = \mathtt{Z}_{i,1}, \ldots, \mathtt{Z}_{i,n_i}$ $(n_i \geq 0)$—i.e., $\overline{\mathtt{Z}}^{(i)}$ are the dependencies of $\mathtt{L}_i$ $(1 \leq i \leq m)$—such that:

(i) $\mathtt{Z}_i \preceq \mathtt{L}_i$ $(1 \leq i \leq m)$, and
(ii) the m DPLs are such that each class can only be declared and modified by at most one of $AB_{\mathtt{L}}, AB_{\mathtt{L}_1}, \ldots, AB_{\mathtt{L}_m}, AB_{\mathtt{Z}'_1}, \ldots AB_{\mathtt{Z}'_k}$ where

$$\mathtt{Z}'_1, \ldots, \mathtt{Z}'_k = ((\bigcup_{1 \leq i \leq m} \overline{\mathtt{Z}}^{(i)}) \setminus \{\mathtt{Z}_1, \ldots, \mathtt{Z}_m\}) \cup \{\mathtt{Z}_{m+1}, \ldots, \mathtt{Z}_n\}\,.$$

Let define the SPL $\mathtt{L}' = \mathcal{N}(\mathtt{L})\bullet\mathcal{R}(\mathcal{N}(\mathtt{L}_1))\bullet \ldots \bullet\mathcal{R}(\mathcal{N}(\mathtt{L}_m))$. The composition of $\mathtt{L}$ with $\overline{\mathtt{L}}$ is the DPL $\mathtt{L}_0$ defined as: **line** $\mathtt{L}_0(\mathtt{Z}'_1, \ldots, \mathtt{Z}'_k)$ $\{$ $\mathcal{M}_{\mathtt{L}'}$ $\mathcal{M}_{Id}$ $\mathcal{K}_{\mathtt{L}'}$ $AB_{\mathtt{L}'}$ $\}$.[4]

Remark 20 (On DPL-DPL and DPL-SPL compositions). Note that, in the previous definition, if we take $m = n$ and the $\overline{\mathtt{L}}$ being SPLs, we obtain the DPL-SPL composition. Hence, the DPL-DPL composition conservatively extends DPL-SPL composition, additionally allowing to: (i) satisfy only a subset of the DPL dependencies; and (ii) satisfy them with DPLs.

[4]Note that in [10], the DPL-DPLs composition is more constrained than in the current definition, with $m = n$ and none of the DPLs $\mathtt{L}_i$ sharing dependencies. The current definition is thus more flexible, as it allows to (i) satisfy only a subset of the DPL dependencies, and (ii) use any SPL or DPL $\mathtt{L}_i$ to satisfy a dependency $\mathtt{Z}_i$, with the only constraint that it must validate the dependency $(\mathtt{Z}_i \preceq \mathtt{L}_i)$ and that there is no name clash. For instance, the DPL-DPLs compositions illustrated in Examples 9 and 10 would not be possible according to the notion of DPL-DPLs composition in [10].

Example 9 (The DepDualAccount DPL). Let consider the DualAccount DPL presented in the Example 6. As suggested in the introduction of this section, this DPL can be refactored in a new DPL, called DepDualAccount, that would contain an additional dependency towards the DLibInt Signature, in order to use its functionalities to record the date of each activity in its logs. The feature model of such DPL is depicted in Figure 12. Following Definition 19 and as suggested in the introduction of this section, it is possible to construct the DPL DepDualAccount(DepCapitalAccount, FinancialAccount): indeed, the DepCapitalAccount DPL and the FinancialAccount SPL respectively implement the dependencies CapAccInt and FinAccInt. The DPL resulting of such composition, called DepCompleteAccount, has thus still one dependency, DLibInt, that is activated either: when the `Log` feature is activated (as stated in the DepDualAccount feature model), or when the `CapitalAccount` feature is activated (as stated in the DepCapitalAccount feature model). The feature model of this DPL is depicted in Figure 13.

It is worth noticing that the result of a DPL-DPL composition $\mathtt{L}(\mathtt{L}_1, \dots, \mathtt{L}_n)$ can be an SPL, even if some of the $\mathtt{L}_i$ are DPLs. This is illustrated in the following example.

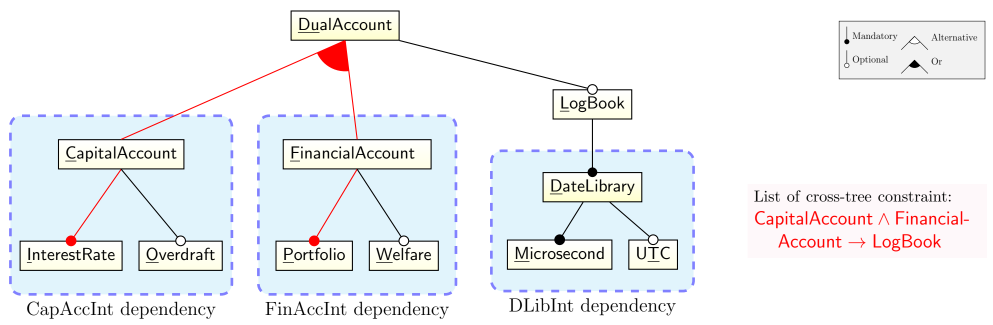
Example 10 (The CompleteDateAccount SPL). Consider the DepDualAccount defined in Example 9, that has three dependencies in total: CapAccInt, FinAccInt and DLibInt. We can note that the result of the DPL-DPL composition DepDualAccount(DepCapitalAccount, FinancialAccount, DateLibrary), that we call CompleteDateAccount, is an SPL even if DepCapitalAccount is a DPL with one dependency. This is due to the fact that the third argument of the composition, DateLibrary, satisfies the dependency of DepCapitalAccount.

4.2.1. Modular DPL-DPLs composition

Like for the DPL-SPLs composition, the more flexible DPL-DPLs composition enjoys some interesting properties for the construction of its feature model and generator. These properties are stated in the two following theorems:

Theorem 5 (Feature Model of the DPL-DPLs composition). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, \dots, \mathtt{Z}_n$*, and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_m$ *($m \leq n$) be DPLs with respective dependencies* $\overline{\mathtt{Z}}^{(i)}$*, such that the composition* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ *is defined. Then, given* $\mathtt{Z}''_1, \dots, \mathtt{Z}''_l = \{\mathtt{Z}_{m+1}, \dots, \mathtt{Z}_n\} \setminus \bigcup_{1\leq i\leq m} \overline{\mathtt{Z}}^{(i)}$*, we have*

$$\mathcal{M}^{Main}_{\mathtt{L}_0} = \mathcal{M}^{Main}_{\mathtt{L}} \bullet \mathcal{M}^{Glue}_{\mathtt{L}} \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_m})) \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{Z}''_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}''_l}))\,.$$



$$
\begin{array}{lll}
\mathcal{F}_{\mathsf{DepDualAccount}} = & \{ & \mathsf{Du\ C\ I\ O\ F\ P\ W\ L\ D\ M\ T}\ \} \\
\mathcal{P}_{\mathsf{DepDualAccount}} = & \{\{ & \mathsf{Du\ C\ I}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ F\ P}\ \}, \\
& \{ & \mathsf{Du\ F\ P\ W}\ \}, \\
& \{ & \mathsf{Du\ F\ P\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ F\ P\ W\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ F\ P\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ F\ P\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ F\ P\ W\ L\ D\ M}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ F\ P\ W\ L\ D\ M}\ \} \\
& \{ & \mathsf{Du\ F\ P\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ F\ P\ W\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ F\ P\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ F\ P\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ F\ P\ W\ L\ D\ M\ T}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ O\ F\ P\ W\ L\ D\ M\ T}\ \}\ \}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{F}^{Main}_{\mathsf{DepDualAccount}} = & \{ & \mathsf{Du\ L}\ \} \\
\mathcal{P}^{Main}_{\mathsf{DepDualAccount}} = & \{\{ & \mathsf{Du}\ \}, \\
& \{ & \mathsf{Du\ L}\ \}\ \}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{F}^{Glue}_{\mathsf{DepDualAccount}} = & \{ & \mathsf{Du\ C\ I\ F\ P\ L\ D}\ \} \\
\mathcal{P}^{Glue}_{\mathsf{DepDualAccount}} = & \{\{ & \mathsf{Du\ C\ I}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ L\ D}\ \}, \\
& \{ & \mathsf{Du\ F\ P}\ \}, \\
& \{ & \mathsf{Du\ F\ P\ L\ D}\ \}, \\
& \{ & \mathsf{Du\ C\ I\ F\ P\ L\ D}\ \}\ \}
\end{array}
$$

Figure 12: $\mathsf{DepDualAccount}$ DPL: feature model $\mathcal{M}_{\mathsf{DepDualAccount}}$.

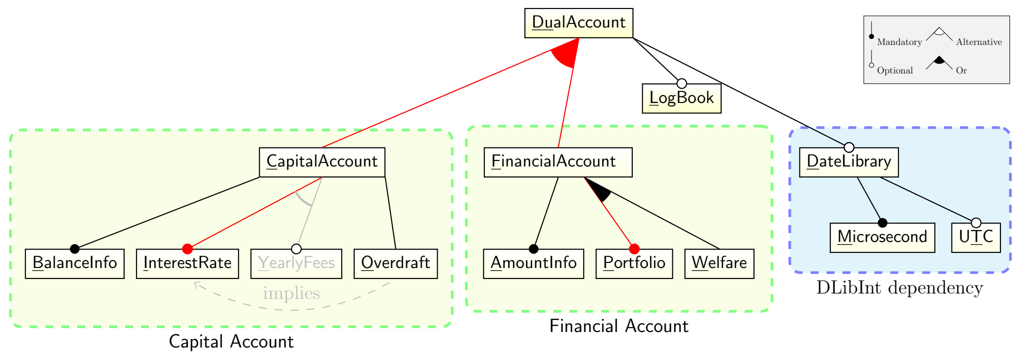
Proof. Similar to that of Theorem 2. □

Theorem 6 (Generator of the DPL-DPLs composition). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ *(*$n \geq 0$*), and* $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_m$ *(*$m \leq n$*) be DPLs such that the composition* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ *is defined. For every product* $p \in \mathcal{P}_{\mathtt{L}_0}$*, we have* $\mathcal{G}_{\mathcal{N}(\mathtt{L}_0)}(p) = \mathcal{G}_{\mathcal{N}(\mathtt{L})}(p \cap \mathcal{F}_{\mathtt{L}}) \cup \left(\bigcup_{\mathtt{L}_i \in \overline{\mathtt{L}}} \mathcal{G}^{\emptyset}_{\mathcal{N}(\mathtt{L}_i)}(p \cap \mathcal{F}_{\mathtt{L}_i})\right)$.

Proof. Similar to that of Theorem 3. □

Additionally, partially fulfilling the dependencies of a DPL with some DPLs still preserves the set of implemented interfaces:

Theorem 7 (Interfaces of DPL-SPLs composition). *Let* $\mathtt{L}$ *be a DPL implementing an SPLS* $\mathtt{Z}$*, with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ *(*$n \geq 0$*). Let*



List of cross-tree constraint:
CapitalAccount ∧ FinancialAccount → LogBook ; Log ∨ CapitalAccount ↔ DateLibrary

$$
\begin{array}{ll}
\mathcal{F}_{\mathsf{DepCompleteAccount}} = & \{\mathsf{Du\ C\ B\ I\ Y\ O\ F\ A\ P\ W\ L\ D\ M\ T}\} \\
\mathcal{P}_{\mathsf{DepCompleteAccount}} = & \{\{\mathsf{Du\ C\ B\ I\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ L\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ L\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ F\ A\ P}\}, \\
& \{\mathsf{Du\ F\ A\ P\ W}\}, \\
& \{\mathsf{Du\ F\ A\ P\ L\ D\ M}\}, \\
& \{\mathsf{Du\ F\ A\ P\ W\ L\ D\ M}\}, \\
& \{\mathsf{Du\ F\ A\ P\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ F\ A\ P\ W\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ F\ A\ P\ L\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ F\ A\ P\ L\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ F\ A\ P\ W\ L\ D\ M}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ F\ A\ P\ W\ L\ D\ M}\} \\
& \{\mathsf{Du\ C\ B\ I\ F\ A\ P\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ F\ A\ P\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ F\ A\ P\ W\ L\ D\ M\ T}\}, \\
& \{\mathsf{Du\ C\ B\ I\ O\ F\ A\ P\ W\ L\ D\ M\ T}\}\}
\end{array}
$$

Figure 13: DepCompleteAccount DPL: feature model $\mathcal{M}_{\mathsf{DepCompleteAccount}}$.

$\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_m$ ($m \leq n$) be DPLs such that the composition $\mathtt{L}(\overline{\mathtt{L}})$ is defined. We then have that $\mathtt{Z} \preceq \mathtt{L}(\overline{\mathtt{L}})$.

Proof. Similar to that of Theorem 4. □

4.3. Compositional analysis of feature models

Schröter et al. [8] proved that many properties (*void feature model*, *core features*, *dead features*, *void partial configuration*, and *atomic sets*) of a feature model composition $\mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_y$ are closely related to any compositions $\mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{Int}$ with $\mathcal{M}_{Int} \preceq \mathcal{M}_y$, if some feature interference constraint is satisfied.[5] For instance, $\mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_y$ is void if and only if $\mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{Int}$ is void. This result can be lifted to DPL-SPL and DPL-DPL compositions to show that the properties of the feature model of a $\mathtt{L}(\overline{\mathtt{L}})$ composition are a direct consequence of the feature model of $\mathtt{L}$ itself.

To state this lifting, we first recall the definition of these properties.

Definition 21 (Feature Model Properties [8]). Given a feature model $\mathcal{M}_x$, we have:

- $\mathcal{M}_x$ is *void* iff $\mathcal{F}_x = \emptyset$.
- the *core features* of $\mathcal{M}_x$, written $core(\mathcal{M}_x)$ is the set of features common to all the products of $\mathcal{M}_x$: $core(\mathcal{M}_x) \triangleq \bigcap_{p \in \mathcal{P}_x} p$.
- the *dead features* of $\mathcal{M}_x$, written $dead(\mathcal{M}_x)$ is the set of features in $\mathcal{F}_x$ that are never used in any product of $\mathcal{M}_x$: $dead(\mathcal{M}_x) \triangleq \mathcal{F}_x \setminus \bigcup_{p \in \mathcal{P}_x} p$.
- the *partial configurations* of $\mathcal{M}_x$, written $pConf(\mathcal{M}_x)$ is the set of pairs of partially chacaterising a product of $\mathcal{M}_x$:
$$pConf(\mathcal{M}_x) \triangleq \bigcup_{p \in \mathcal{P}_x} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \mathcal{F}_S \subseteq p, \mathcal{F}_D \subseteq \mathcal{F}_x \setminus p\}$$
- the *atomic subsets* of $\mathcal{M}_x$, written $aSub(\mathcal{M}_x)$, is the set of non-empty sets of features which are either completely included or completely absent in each product:
$$aSub(\mathcal{M}_x) \triangleq \begin{cases} \emptyset & \text{if } \mathcal{P}_x = \emptyset \\ \{q \mid q \subseteq \mathcal{F}_x, \forall p \in \mathcal{P}_x,\ q \cap p \neq \emptyset \Leftrightarrow q \cap p = q\} & \text{else} \end{cases}$$

The following theorem states that the results shown in [8] can be lifted to DPL-DPLs composition. It is straightforward to check that the same is true for the DPL-SPLs composition.

Theorem 8 (Compositional analysis of feature models). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, \ldots, \mathtt{Z}_n$ $(n \geq 0)$*, and* $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_n$ *be DPLs such that:*

[5]More precisely, if all the feature dependencies in the composition with $\mathcal{M}_y$ are captured by the composition with $\mathcal{M}_{Int}$, i.e., $(\mathcal{F}_x \cup \mathcal{F}_{Glue}) \cap \mathcal{F}_y = (\mathcal{F}_x \cup \mathcal{F}_{Glue}) \cap \mathcal{F}_{Int}$.

(i) $\mathtt{L}' = \mathtt{L}(\overline{\mathtt{L}})$ *is defined;*
(ii) *for all* $1 \leq i \leq n$ *we have* $\mathcal{F}_{\mathtt{L}_i} \cap (\mathcal{F}_{\mathtt{L}} \cup \bigcup_{j \neq i} \mathcal{F}_{\mathtt{L}_j}) \subset \mathcal{F}_{\mathtt{Z}_i}$ *(which corresponds to the hypothesis (3) of Lemma 13 extended to several feature models).*

Then the following properties hold:

(i) $\mathcal{M}_{\mathtt{L}'}$ *is void iff* $\mathcal{M}_{\mathtt{L}}$ *is void;*
(ii) $core(\mathcal{M}_{\mathtt{L}}) = (core(\mathtt{L}') \cap \mathcal{F}_{\mathtt{L}})$*;*
(iii) $dead(\mathcal{M}_{\mathtt{L}}) = (dead(\mathtt{L}') \cap \mathcal{F}_{\mathtt{L}})$*;*
(iv) $pConf(\mathcal{M}_{\mathtt{L}}) = \{(\mathcal{F}_S \cap \mathcal{F}_{\mathtt{L}}, \mathcal{F}_D \cap \mathcal{F}_{\mathtt{L}}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_{\mathtt{L}'})\}$*;*
(v) $aSub(\mathcal{M}_{\mathtt{L}}) = \{q \cap \mathcal{F}_{\mathtt{L}} \mid a \in aSub(\mathcal{M}_{\mathtt{L}'}) \wedge q \cap \mathcal{F}_{\mathtt{L}} \neq \emptyset\}$.

Proof. See Appendix E. □

Example 11 (Compositional analysis of feature models). The DPL-SPL composition in Example 7 validates the hypothesis of Theorem 8. However, this is not the case of DPL-DPL composition in Example 9. Indeed, because the dependency toward the SPLS DLibInt is shared between the DPLs DepCapitalAccout and DepDualAccount, the hypothesis (ii) of the theorem is not validated. We can additionally remark that the equality on *pConf* stated in point (iv) of the theorem is not true between DepDualAccount and DepCompleteAccount: while $(\{\mathtt{Du}, \mathtt{C}, \mathtt{I}\}, \{\mathtt{D}, \mathtt{C}\})$ is in $pConf(\mathcal{M}_{\mathsf{DepDualAccount}})$, no corresponding partial product is in $pConf(\mathcal{M}_{\mathsf{DepCompleteAccount}})$.

4.4. Compositional type checking for MPLs

In this section we show that composing type safe DPLs yields a type safe DPL (Definition 22 and Theorem 9). This implies that type safety of an DPL resulting from a DPL-DPLs composition (or an SPL resulting from a DPL-SPLs composition) can be checked by type checking in isolation the SPLs and DPLs involved into the composition by using any (sound)[6] type system for SPLs (like, e.g., the type systems in [9, 16, 17]). Note that, if the type system is not complete,[7] it might not be able to type check the composed DPL which is nonetheless guaranteed to be type safe.

To introduce this result, we first give a formal definition of type safety for DPL.

[6]A type system for SPLs is *sound* if all the SPL that can be typed by it are type safe.
[7]A type system for SPLs is *complete* if all the type safe SPLs can be typed by it.

Definition 22 (Type safe DPL). A DPL $\mathtt{L}$ with dependencies $\mathtt{Z}_1, \ldots \mathtt{Z}_n$ is type safe iff the SPL $\mathtt{L}(\mathtt{Z}_1^{\bigstar}, \ldots, \mathtt{Z}_n^{\bigstar})$ is type safe.

Note that Definition 22 generalizes Definition 5: when the set of dependencies of the DPL $\mathtt{L}$ is empty ($n = 0$), $\mathtt{L}$ is type safe if and only if it is type safe in the SPL-sense of the term. Moreover, with this definition, extending the existing type-checking algorithms for SPL to manage DPLs is direct: it simply requires to first extend the input DPL with the configuration knowledges and the stub-completed deltas of its dependencies (thus computing $\mathtt{L}(\mathtt{Z}_1^{\bigstar}, \ldots, \mathtt{Z}_n^{\bigstar})$), and second to apply the type checking algorithm to the resulting SPL.

The following theorem states that DPL-DPLs composition preserves type safety, the analogous result for DPL-SPLs composition can be straightforwardly derived from it.

Theorem 9 (DPL-DPLs composition preserves type safety). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\mathtt{Z}_1, \ldots, \mathtt{Z}_n$ *(*$n \geq 0$*), and* $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_m$ *(*$m \leq n$*) be DPLs such that* $\mathtt{L}(\overline{\mathtt{L}})$ *is defined. If each of the DPLs* $\mathtt{L}, \mathtt{L}_1, \ldots, \mathtt{L}_m$ *is type safe, then* $\mathtt{L}(\overline{\mathtt{L}})$ *is type safe.*

Proof. See Appendix F. □

Example 12 (Type safety of the running example). Theorem 9 can be used to check the type safety of all the SPL compositions presented in our examples. Indeed, the attentive reader can check that the SPLs CapitalAccount (presented in Example 1) and FinancialAccount (presented in Example 3) are type safe. Moreover, as the DualAccount DPL (presented in Example 6) is also type safe, we can conclude that the SPL CompleteAccount (presented in Example 7) is type safe.

Also, considering that the refactored code of the DepCapitalAccount DPL (presented in Example 8) and the DepDualAccount DPL (presented in Example 9) are type safe, we can also conclude that the DPL DepCompleteAccount (presented in Example 9) and the SPL CompleteDateAccount (presented in Example 10) are type safe as well.

5. Checking the SPL interface relation

Due to the variable nature of SPLSs, checking the subsignature relation is challenging. Indeed, a direct approach that iterates over all the variants of the SPLSs is not feasible when the number of features is large (a product line with n features can have up to 2^n variants).

In this section, following the approach taken by the authors in [17] for type checking delta-oriented SPLs, we show how to encode subinterface checking into SAT satisfiability (which is a co-NP problem). We thus provide a subinterface checking algorithm which can take advantage of the many heuristics implemented in SAT solvers (a SAT solver can be used to check the satisfiability of a propositional formula by checking if its negation is unsatisfiable): similar translations into SAT constraints have been applied in practice for several SPL analysis with good results [18, 19, 20]. Our approach is based on following functions returning SAT constraints over features:

(i) $\mathfrak{P}(\mathtt{Z})$ gives the constraint describing the products of $\mathtt{Z}$ (i.e., a propositional formula over features describing $\mathcal{P}_{\mathtt{Z}}$).
(ii) $\mathfrak{C}(\mathtt{Z}, \mathtt{C})$ gives the constraint describing the products of $\mathtt{Z}$ whose respective variants contain the class $\mathtt{C}$.
(iii) $\mathfrak{I}(\mathtt{Z}, \mathtt{C}, \mathtt{C}')$ gives the constraint describing the products of $\mathtt{Z}$ whose respective variants have $\mathtt{C}$ being a subtype of $\mathtt{C}'$.
(iv) $\mathfrak{L}(\mathtt{Z}, \mathtt{C}, \mathtt{a}, \mathit{AS})$ gives the constraint describing the products of $\mathtt{Z}$ whose respective variants PS are such that $\mathit{lookup}_{\mathit{PS}}(\mathtt{a}, \mathtt{C})$ is defined and is equal to AS.

Additionally, we use the two following getters:

(i) $\mathtt{gclass}(\mathtt{Z})$ returns the set of class names declared in the artifact base of $\mathtt{Z}$.
(ii) $\mathtt{gatt}(\mathtt{Z}, \mathtt{C})$ returns a set of pairs $(\mathtt{a}, \mathit{AS})$ such that either the base program or a delta in the artifact base of $\mathtt{Z}$ defines $\mathtt{a}$ as an attribute of $\mathtt{C}$, with the declaration AS.

The complete definition of these functions and getters is given in Appendix G. The principle of our encoding is the following. Consider checking that $\mathtt{Z}_1 \preceq \mathtt{Z}_2$ and suppose that $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$: we generate a constraint that expresses that for all the products p of $\mathtt{Z}_2$, the three points of subinterfacing for program signatures are validated between the variants $\mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1})$ and $\mathcal{G}_{\mathtt{Z}_2}(p)$. This constraint is defined as follows:

$$\begin{array}{rl}\mathfrak{S}(\mathtt{Z}_1,\mathtt{Z}_2) \triangleq \mathfrak{P}(\mathtt{Z}_2) \Rightarrow (& \\ & \bigwedge_{\mathtt{C}}(\mathfrak{C}(\mathtt{Z}_1,\mathtt{C}) \Rightarrow \mathfrak{C}(\mathtt{Z}_2,\mathtt{C})) \\ & \wedge \bigwedge_{\mathtt{C}} \bigwedge_{\mathtt{C}'}(\mathfrak{I}(\mathtt{Z}_1,\mathtt{C},\mathtt{C}') \Rightarrow \mathfrak{I}(\mathtt{Z}_2,\mathtt{C},\mathtt{C}')) \\ & \wedge \bigwedge_{\mathtt{C}} \bigwedge_{\mathtt{a},AS}(\mathfrak{L}(\mathtt{Z}_1,\mathtt{C},\mathtt{a},AS) \Rightarrow \mathfrak{L}(\mathtt{Z}_2,\mathtt{C},\mathtt{a},AS))\ ) \\ & \qquad\qquad \text{with } \left\{ \begin{array}{l} \mathtt{C},\mathtt{C}' \in \mathtt{gclass}(\mathtt{Z}_1) \\ \text{and } (\mathtt{a},AS) \in \mathtt{gatt}(\mathtt{Z}_1,\mathtt{C}) \end{array} \right. \end{array}$$

In the first line of this contraint, $\mathfrak{P}(\mathtt{Z}_2)$ acts as an iterator over all the products p of $\mathtt{Z}_2$. In the second line, $\mathfrak{C}(\mathtt{Z}_1,\mathtt{C}) \Rightarrow \mathfrak{C}(\mathtt{Z}_2,\mathtt{C})$ checks that all the classes declared in $\mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1})$ are also declared in $\mathcal{G}_{\mathtt{Z}_2}(p)$. Note that this constraint works because we have $\mathcal{P}_{\mathtt{Z}_1} = \{p \cap \mathcal{F}_{\mathtt{Z}_1} \mid p \in \mathcal{P}_{\mathtt{Z}_2}\}$: that way, when $\mathfrak{P}(\mathtt{Z}_2)$ selects a product p of $\mathtt{Z}_2$, it implicitly selects the corresponding product $p \cap \mathcal{F}_{\mathtt{Z}_1}$ of $\mathtt{Z}_1$. In the third line, $\mathfrak{I}(\mathtt{Z}_1,\mathtt{C},\mathtt{C}') \Rightarrow \mathfrak{I}(\mathtt{Z}_2,\mathtt{C},\mathtt{C}')$ ensures that all inheritance relations declared in $\mathcal{G}_{\mathtt{Z}_1}(p\cap\mathcal{F}_{\mathtt{Z}_1})$ are also declared in $\mathcal{G}_{\mathtt{Z}_2}(p)$. And Finally, in the fourth line, $\mathfrak{L}(\mathtt{Z}_1,\mathtt{C},\mathtt{a},AS) \Rightarrow \mathfrak{L}(\mathtt{Z}_2,\mathtt{C},\mathtt{a},AS)$ checks that for all $\mathtt{C}$ and $\mathtt{a}$ such that $\mathit{lookup}_{\mathcal{G}_{\mathtt{Z}_1}(p\cap\mathcal{F}_{\mathtt{Z}_1})}(\mathtt{a},\mathtt{C})$ is defined, we have that $\mathit{lookup}_{\mathcal{G}_{\mathtt{Z}_1}(p\cap\mathcal{F}_{\mathtt{Z}_1})}(\mathtt{a},\mathtt{C}) = AS = \mathit{lookup}_{\mathcal{G}_{\mathtt{Z}_2}(p)}(\mathtt{a},\mathtt{C})$.

The soundness and completeness of the constraint $\mathfrak{S}(\mathtt{Z}_1,\mathtt{Z}_2)$ is stated in the following theorem.

Theorem 10 (SPL interface checking). *Suppose given two SPL signatures $\mathtt{Z}_1$ and $\mathtt{Z}_2$ with $\mathcal{G}_{\mathtt{Z}_1}$ and $\mathcal{G}_{\mathtt{Z}_2}$ total. Then the two following statements are equivalent:*

(i) $\mathtt{Z}_1 \preceq \mathtt{Z}_2$

(ii) $\mathcal{F}_{\mathtt{Z}_1} \subseteq \mathcal{F}_{\mathtt{Z}_2}$, $(\underset{f\in\mathcal{F}_{\mathtt{Z}_2}\setminus\mathcal{F}_{\mathtt{Z}_1}}{\exists} f.\mathfrak{P}(\mathtt{Z}_2)) \Leftrightarrow \mathfrak{P}(\mathtt{Z}_1)$ *and* $\mathfrak{S}(\mathtt{Z}_1,\mathtt{Z}_2)$ *are valid.*

Proof. See Appendix G. □

6. Related work

Schröter et al. [21] advocated the idea of using *dependent* feature models to build MPLs, and explore several analysis of such feature model. Then, in [22], they advocated investigating suitable interfaces in order to support compositional analyses of MPLs for different stages of the development process. In particular, they informally introduced the notion of *syntactical interface*, which build on feature model interfaces to provide a view of reusable programming artifacts, and the notion of *behavioral interfaces*, which in turn build on syntactical interfaces to support formal verification. More recently, Schröter et al. [8] provided a formal account of some of the ideas outlined in [21]. They proposed a concept of feature model interface that consists of a subset of features (thus it hides all other features and dependencies) and used it in combination with a concept of feature model composition through aggregation to support compositional analyses of feature models—see Section 2.2. In this paper we propose the concepts of SPLS, DPL, and DPL-DPLs composition and show how to use them to support compositional type checking of delta-oriented MPLs. Our notion of DPL-DPLs encompasses the notions of feature model interface and feature model composition proposed in [8], while our notion of SPL signature is (according to terminology of [22]) a syntactical interface that provides a variability-aware API, expressed in the flexible and modular DOP approach, specifying which classes and members of the variants of a DPL are intended to be accessible by variants of other DPLs.

Feature-context interfaces [23] are aimed at preventing type errors in SPLs developed according to the FOP approach which, as pointed out in Section 1, is encompassed by DOP (see [24] for a detailed comparison between FOP and DOP). A feature-context interface supports preventing type errors in the context of a set of features *FC*. It provides an invariable API specifying classes and members of the feature modules corresponding to the features in *FC* that are intended to be accessible. In contrast, our concept of SPLS represents a variability-aware API that supports compositional type checking of MPLs. Notably, since DOP is an extension of FOP, our results apply also to FOP SPLs.

Kästner et al. [25] proposed a variability-aware module system, where each module represents an SPL, that allows for type checking modules in isolation. Variability inside each module and its interface is expressed by means of `#ifdef` preprocessor directives and variable linking, respectively. In contrast to our SPLSs, module interfaces do not support hiding features and dependencies. A major difference with respect to our proposal is in the approach used to implement variability (i.e., to build variants): [25] considers an *annotative approach* (`#ifdef` preprocessor directives), while we consider a *transformational approach* (DOP)—we refer to [26, 20] for classification and survey of different approaches for implementing variability.

7. Conclusion and future work

In this paper, we introduced a formal model of MPLs for Delta-Oriented Programming, spanning feature model, artifact base and configuration knowledge, and based on the notions of SPL, SPL signature and DPL. Using these different notions, we defined two composition operators on SPL s and DPLs, that allow to construct complex SPLs and DPLs from simpler ones. We also demostrated several properties of these composition operators. We illustrated how the feature model and variants of the composed SPL or DPL can be computed without actually performing the composition. We lifted the composition properties demonstrated on feature models in [8] to SPL and our composition mechanisms. And we proved that type checking is compositional in our framework, i.e., to ensure the type safety of a complex composition of many SPLs and DPLs, it is enough to type check each SPL and DPL individually.

We plan to integrate our approach to support MPL into DeltaJ 1.5 [27, 28] (a prototypical implementation of DOP that supports full Java 1.5) and into the Abstract Behavioral Specification (ABS) delta-oriented modeling language and toolchain [29, 30]. DeltaFineFit [31] is a recently proposed model-based testing approach that lifts to DeltaJ 1.5 SPLs the FineFit [32] model-based testing approach of Java programs—both FineFit and DeltaFineFit rely on the notion of *data refinement* [33] to compare the state of the model with the state of the *system under test* (SUT). In future work we would like to explore the possibility of extending the DeltaFineFit approach to MPLs. ABS is currently used by an industrial project with German Railways (DB Netz AG) [34] that includes a detailed, executable model of railway infrastructure—in this domain the notion of MPL naturally arises [35, 36]. Implementing our approach as part of the ABS toolchain and applying it on industrial case studies should allow us to evaluate its practical usefulness (e.g., how this theory helps the system designer) and the scalability of the SPL interface checking algorithm.

In future work we would like to investigate the possibility of designing a generic framework that supports MPLs that comprise SPL that adopt different representations for the feature model, SPL interfaces and variants. A preliminary step in this direction is in illustrated in [37] which presents an MPL model that adopts a generic notion of variant and replaces the notions of SPL signature and interface with a more flexible notion of subtyping. The model is first illustrated by encoding a fragment of an MPL (developed with the toolchain of the HyVar project [38, 39]) comprising delta-oriented SPLs where variants are statecharts [40] expressed in the format supported by Yakindu Statechart Tools [41]. Then the model is used to encode *Portage* [42] (the package manager of the *Gentoo* [43] Linux distribution) and to extract statistical information from it.

Recently, Thüm et al. [44] proposed a notion of behavioral interface (i.e., a variability-aware contract) for supporting compositional verification of FOP SPLs via variability encoding [45]. In particular, they investigated how variability hiding (i.e., the interface relation between feature models) can help contract checking upon SPL evolution, like it does for type checking. In future work we would like to enrich SPLSs with method contracts (thus promoting them to behavioral interfaces) in order to support compositional verification of delta-oriented DPLs by building on recently proposed proof systems and techniques for the verification of delta-oriented SPLs [46, 47, 48, 49, 50].

Acknowledgments

We thank the anonymous FSEN 2017 reviewers for comments and suggestions on the version of this paper submitted to FSEN 2017, Lorenzo Testa for comments and suggestions during the preparation of the post-proceedings version, and the anonymous SCP reviewers for many insightful comment and suggestions for improving the presentation.

Appendix A. The join operation on feature models is associative and commutative

The proof relies on the following preliminary lemma.

Lemma 11. *Let $\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i)$ be feature models and $p_i \in \mathcal{P}_i$ $(1 \leq i \leq 3)$.*

1. *If $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ and $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ then $p_1 \cap \mathcal{F}_2 = p_2 \cap \mathcal{F}_1$.*
2. *If $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ and $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ then $p_1 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_1$.*
3. *If $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ and $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ then $p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2) = (p_1 \cup p_2) \cap \mathcal{F}_3$.*

Proof. 1. We start by proving $p_1 \cap \mathcal{F}_2 \subseteq p_2 \cap \mathcal{F}_1$. If $f \in p_1 \cap \mathcal{F}_2$ then $f \in p_1$, $f \in \mathcal{F}_2$ and also $f \in p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3)$. But $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) =$

$(p_2 \cup p_3) \cap \mathcal{F}_1$ by hypothesis, thus $f \in (p_2 \cup p_3) \cap \mathcal{F}_1$ and, in particular $f \in p_2 \cup p_3$, $f \in \mathcal{F}_1$. There are two cases. First, $f \in p_2$. Second, $f \in p_3$ so $f \in p_3 \cap \mathcal{F}_2$; but $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ by hypothesis, thus $f \in p_2 \cap \mathcal{F}_3$ and, in particular, $f \in p_2$. In both cases $f \in p_2 \cap \mathcal{F}_1$.
We conclude by proving that $p_2 \cap \mathcal{F}_1 \subseteq p_1 \cap \mathcal{F}_2$. If $f \in p_2 \cap \mathcal{F}_1$ then $f \in p_2$, $f \in \mathcal{F}_1$ and also $f \in (p_2 \cup p_3) \cap \mathcal{F}_1$, $f \in \mathcal{F}_2$. But $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ by hypothesis, so $f \in p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3)$ and, in particular, $f \in p_1$. So, $f \in p_1 \cap \mathcal{F}_2$.

2. We start by proving $p_1 \cap \mathcal{F}_3 \subseteq p_3 \cap \mathcal{F}_1$. If $f \in p_1 \cap \mathcal{F}_3$ then $f \in \mathcal{F}_1$, $f \in p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3)$ and by hypothesis, $f \in (p_2 \cup p_3) \cap \mathcal{F}_1$. There are two cases. First, $f \in p_2$ so $f \in p_2 \cap \mathcal{F}_3$; by hypothesis $f \in p_3 \cap \mathcal{F}_2$, thus $f \in p_3$ and $f \in p_3 \cap \mathcal{F}_1$. Second, $f \in p_3$ and immediately $f \in p_3 \cap \mathcal{F}_1$. In both cases the inclusion is proved.
We conclude by proving that $p_3 \cap \mathcal{F}_1 \subseteq p_1 \cap \mathcal{F}_3$. If $f \in p_3 \cap \mathcal{F}_1$ then $f \in p_3$, $f \in \mathcal{F}_1$ and also $f \in \mathcal{F}_3$ and $f \in (p_2 \cup p_3) \cap \mathcal{F}_1$. By hypothesis $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ thus $f \in p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3)$. Therefore $f \in p_1$ and $f \in p_1 \cap \mathcal{F}_3$.
3. We start by proving $p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2) \subseteq (p_1 \cup p_2) \cap \mathcal{F}_3$. If $f \in p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2)$ then $f \in p_3$, $f \in (\mathcal{F}_1 \cup \mathcal{F}_2)$ and also $f \in \mathcal{F}_3$. There are two cases. First $f \in \mathcal{F}_1$, so $f \in p_3 \cap \mathcal{F}_1$; but $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$ by hypothesis, thus $f \in p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3)$ and, in particular, $f \in p_1$. Therefore, $f \in p_1 \cap \mathcal{F}_3 \subseteq (p_1 \cup p_2) \cap \mathcal{F}_3$. Second, $f \in \mathcal{F}_2$, so $f \in p_3 \cap \mathcal{F}_2$; but $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ by hypothesis, thus $f \in p_2 \cap \mathcal{F}_3$ and, in particular, $f \in p_2$. Therefore, $f \in p_2 \cap \mathcal{F}_3 \subseteq (p_1 \cup p_2) \cap \mathcal{F}_3$.
We conclude by proving that $(p_1 \cup p_2) \cap \mathcal{F}_3 \subseteq p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2)$. If $f \in (p_1 \cup p_2) \cap \mathcal{F}_3$ then $f \in p_1 \cup p_2$, $f \in \mathcal{F}_3$. There are two case. First, if $f \in p_1$ then $f \in p_1 \cap \mathcal{F}_3$. But by Lemma 11.2, $p_1 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_1$, thus $f \in p_3 \cap \mathcal{F}_1 \subseteq p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2)$. Second, if $f \in p_2$ then $f \in p_2 \cap \mathcal{F}_3$. But by hypothesis, $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$, thus $f \in p_3 \cap \mathcal{F}_2 \subseteq p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2)$. Concluding, in both cases the proof follows. □

Proposition 12 (Join operation on feature models). *The join operation on feature models* $\bullet$ *is associative and commutative, with* $\mathcal{M}_{Id} = \mathcal{R}(\mathcal{M}_\emptyset) = \mathcal{R}((\emptyset, \emptyset)) = (\emptyset, \{\emptyset\})$ *as identity.*

Proof. Recall that

$$\mathcal{M}_x \bullet \mathcal{M}_y = (\mathcal{F}_x \cup \mathcal{F}_y, \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}).$$

The commutativity of $\bullet$ easily follows from the commutativity of the set-theoretical union. The fact that $\mathcal{M}_{Id}$ is the $\bullet$-identity is straightforward. On the other hand, the associativity is non-trivial at all, so its proof details follow. Let $\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i)$ be feature models. We aim to prove that $\mathcal{M}_1 \bullet (\mathcal{M}_2 \bullet \mathcal{M}_3) \subseteq (\mathcal{M}_1 \bullet \mathcal{M}_2) \bullet \mathcal{M}_3$. Let $p_1 \cup (p_2 \cup p_3) \in \mathcal{M}_1 \bullet (\mathcal{M}_2 \bullet \mathcal{M}_3)$, thus $p_2 \cap \mathcal{F}_3 = p_3 \cap \mathcal{F}_2$ and $p_1 \cap (\mathcal{F}_2 \cup \mathcal{F}_3) = (p_2 \cup p_3) \cap \mathcal{F}_1$. Point 1 of Lemma 11 implies $p_1 \cap \mathcal{F}_2 = p_2 \cap \mathcal{F}_1$, so $p_1 \cup p_2 \in \mathcal{P}_{\mathcal{M}_1 \bullet \mathcal{M}_2}$. Point 3 of Lemma 11 implies $p_3 \cap (\mathcal{F}_1 \cup \mathcal{F}_2) = (p_1 \cup p_2) \cap \mathcal{F}_3$, so $p_1 \cup p_2 \cup p_3 \in \mathcal{P}_{(\mathcal{M}_1 \bullet \mathcal{M}_2) \bullet \mathcal{M}_3}$.
The proof of the other inclusion is similar. $\square$

Appendix B. Proof of Lemma 1 and Theorem 2

Proof (of Lemma 1). The first property is direct. Let us write $\mathcal{M}_w = \mathcal{R}(\mathcal{M}_y)$ and $\mathcal{M}_z = \mathcal{R}(\mathcal{M}_x)$: we have $\mathcal{F}_w = \mathcal{F}_y \subseteq \mathcal{F}_x = \mathcal{F}_z$ and

$$\begin{aligned}\mathcal{P}_w &= \mathcal{P}_y \cup \{\emptyset\} = \{p \cap \mathcal{F}_y \mid p \in \mathcal{P}_x\} \cup \{\emptyset\} \\ &= \{p \cap \mathcal{F}_y \mid p \in \mathcal{P}_x \cup \{\emptyset\}\} = \{p \cap \mathcal{F}_w \mid p \in \mathcal{P}_z\}\end{aligned}$$

For the second property, Let us write $\mathcal{M}_j = \mathcal{M}_x \bullet \mathcal{M}_y$. By definition of the $\bullet$ operator, we have $\mathcal{F}_j = \mathcal{F}_x \cup \mathcal{F}_y = \mathcal{F}_x$. Moreover, we have

$$\begin{aligned}\mathcal{P}_j &= \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\} \\ &= \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q\} = \{p \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q\} \\ &\subseteq \mathcal{P}_x\end{aligned}$$

Finally, consider $p \in \mathcal{P}_x$: by definition of $\mathcal{M}_y$, there exists $p' \in \mathcal{P}_y$ such that $p' = p \cap \mathcal{F}_y$. Hence $p \in \mathcal{P}_j$, which implies $\mathcal{P}_x \subseteq \mathcal{P}_j$. $\square$

Proof (of Theorem 2). This is a direct consequence of the definition of $\mathcal{M}_{\mathtt{L}_0}$ and of Proposition 12 and Lemma 1. We can first remark that because $\mathtt{L}_0$ is defined, we have that $\mathtt{Z}_i \preceq \mathtt{L}_i$ for all $1 \leq i \leq n$. By Lemma 1, we thus have that $\mathcal{R}(\mathtt{Z}_i) \preceq \mathcal{R}(\mathtt{L}_i)$ for all $1 \leq i \leq n$. We can then conclude:

$$\begin{aligned}\mathcal{M}_{\mathtt{L}_0} &= (\mathcal{M}_{\mathtt{L}}^{Main} \circ_{\mathcal{M}_{\mathtt{L}}^{Glue}} (\mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n})) \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})) \\ &= (\mathcal{M}_{\mathtt{L}}^{Main} \bullet \mathcal{M}_{\mathtt{L}}^{Glue} \bullet \mathcal{R}(\mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n})) \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})) \\ &= (\mathcal{M}_{\mathtt{L}}^{Main} \bullet \mathcal{M}_{\mathtt{L}}^{Glue} \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n})) \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})) \\ &\overset{\text{(Proposition 12)}}{=} \mathcal{M}_{\mathtt{L}}^{Main} \bullet \mathcal{M}_{\mathtt{L}}^{Glue} \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_1})) \bullet \cdots \bullet (\mathcal{R}(\mathcal{M}_{\mathtt{Z}_n}) \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})) \\ &\overset{\text{(Lemma 1)}}{=} \mathcal{M}_{\mathtt{L}}^{Main} \bullet \mathcal{M}_{\mathtt{L}}^{Glue} \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})\end{aligned}$$

$\square$

Appendix C. Proof of Theorem 3

Proof (of Theorem 3). Let show that the delta activated by the product p are the same in $\mathcal{G}_{\mathtt{L}_0}(p)$ and $\mathcal{G}_{\mathcal{N}(\mathtt{L})}(p\cap\mathcal{F}_{\mathtt{L}})\cup\big(\bigcup_{\mathtt{L}_i\in\bar{\mathtt{L}}}\mathcal{G}^{\emptyset}_{\mathtt{L}_i}(p\cap\mathcal{F}_{\mathtt{L}_i})\big)$. This property is clear from the definition of $\alpha_{\mathtt{L}_0}$. Then, the delta can be applied in the same order, as $<_{\mathtt{L}_0}$ is the union of the orders $<_{\mathtt{L}}$ and $<_{\mathcal{R}(\mathtt{L}_i)}$. □

Appendix D. Proof of Theorem 4

Proof (of Theorem 4). Let write $\mathtt{L}_0 = \mathtt{L}(\bar{\mathtt{L}})$. Following Definition 10, we need to prove that (*i*) $\mathcal{M}_{\mathtt{Z}} \preceq \mathcal{M}_{\mathtt{L}_0}$ and that (*ii*) for all product $p \in \mathcal{P}_{\mathtt{L}_0}$, we have that $\mathcal{G}_{\mathtt{Z}}(p\cap\mathcal{F}_{\mathtt{Z}}) \preceq \mathcal{G}_{\mathtt{L}_0}(p)$.

(i) We can first remark that $\mathcal{F}_{\mathtt{Z}} \subseteq \mathcal{F}_{\mathtt{L}_0}$, as $\mathcal{F}_{\mathtt{Z}} \subseteq \mathcal{F}_{\mathtt{L}}$ and $\mathcal{F}_{\mathtt{L}} \subseteq \mathcal{F}_{\mathtt{L}_0}$. Let now consider $p \in \mathcal{P}_{\mathtt{L}_0}$: by construction, we have that $p\cap\mathcal{F}_{\mathtt{L}} \in \mathcal{P}_{\mathtt{L}}$. Hence, as $\mathtt{Z} \preceq \mathtt{L}$, we have that $p\cap\mathcal{F}_{\mathtt{Z}} = (p\cap\mathcal{F}_{\mathtt{L}})\cap\mathcal{F}_{\mathtt{Z}} \in \mathcal{P}_{\mathtt{Z}}$.

(ii) Let now consider $p \in \mathcal{P}_{\mathtt{L}_0}$:

$$\begin{aligned}
\mathcal{G}_{\mathtt{Z}}(p\cap\mathcal{F}_{\mathtt{Z}}) &\overset{(\text{by } \mathtt{Z}\preceq\mathtt{L})}{\preceq} \mathcal{G}_{\mathtt{L}}(p\cap\mathcal{F}_{\mathtt{L}}) \\
&\preceq \mathcal{G}_{\mathtt{L}}(p\cap\mathcal{F}_{\mathtt{L}}) \cup \big(\textstyle\bigcup_i \mathcal{G}^{\emptyset}_{\mathtt{L}_i}(p\cap\mathcal{F}_{\mathtt{L}_i})\big) \\
&\overset{(\text{Theorem 3})}{=} \mathcal{G}_{\mathtt{L}_0}(p)
\end{aligned}$$

□

Appendix E. Proof of Theorem 8

Lemma 13 (Feature model interface and join). *Let $\mathcal{M}_x$, $\mathcal{M}_{x'}$, $\mathcal{M}_y$ and $\mathcal{M}_{y'}$ be four feature models such that*

$$\mathcal{M}_{x'} \preceq \mathcal{M}_x \quad (1) \qquad \mathcal{M}_{y'} \preceq \mathcal{M}_y \quad (2) \qquad \mathcal{F}_x\cap\mathcal{F}_y = \mathcal{F}_{x'}\cap\mathcal{F}_{y'} \quad (3)$$

Then we have $\mathcal{M}_{x'}\bullet\mathcal{M}_{y'} \preceq \mathcal{M}_x\bullet\mathcal{M}_y$.

Proof. We first remark that, using the hypothesis (1) and (2), we have $\mathcal{F}_{x'} \subseteq \mathcal{F}_x$ and $\mathcal{F}_{y'} \subseteq \mathcal{F}_y$, which implies, using the hypothesis (3) that

$$\mathcal{F}_x\cap\mathcal{F}_y = \mathcal{F}_{x'}\cap\mathcal{F}_{y'} = \mathcal{F}_x\cap\mathcal{F}_{y'} = \mathcal{F}_{x'}\cap\mathcal{F}_y$$

Let now write $\mathcal{M}_z = \mathcal{M}_x \bullet \mathcal{M}_y$ and $\mathcal{M}_{z'} = \mathcal{M}_{x'} \bullet \mathcal{M}_{y'}$. We first have that $\mathcal{F}_{z'} = \mathcal{F}_{x'} \cup \mathcal{F}_{y'} \subseteq \mathcal{F}_x \cup \mathcal{F}_y = \mathcal{F}_z$. Then, with $p \in \mathcal{P}_x$ (and similarily with $q \in \mathcal{P}_y$), we can note that:

$$\begin{array}{rcl} (\mathcal{F}_{y'} \setminus \mathcal{F}_{x'}) \cap p & \subseteq & (\mathcal{F}_{y'} \setminus \mathcal{F}_{x'}) \cap \mathcal{F}_x \\ & \subseteq & (\mathcal{F}_{y'} \cap \mathcal{F}_x) \setminus \mathcal{F}_{x'} \\ & \subseteq & (\mathcal{F}_{y'} \cap \mathcal{F}_{x'}) \setminus \mathcal{F}_{x'} \subseteq \emptyset \end{array}$$

Consequently, we have:

$$\begin{array}{l} (p \cup q) \cap (\mathcal{F}_{x'} \cup \mathcal{F}_{y'}) \\ \quad = \quad (p \cap (\mathcal{F}_{x'} \cup \mathcal{F}_{y'})) \cup (q \cap (\mathcal{F}_{x'} \cup \mathcal{F}_{y'})) \\ \quad = \quad (p \cap \mathcal{F}_{x'}) \cup (p \cap (\mathcal{F}_{y'} \setminus \mathcal{F}_{x'})) \cup (q \cap \mathcal{F}_{y'}) \cup (q \cap (\mathcal{F}_{x'} \setminus \mathcal{F}_{y'})) \\ \quad = \quad (p \cap \mathcal{F}_{x'}) \cup \emptyset \cup (q \cap \mathcal{F}_{y'}) \cup \emptyset \\ \quad = \quad (p \cap \mathcal{F}_{x'}) \cup (q \cap \mathcal{F}_{y'}) \end{array}$$

We then can conclude:

$$\begin{array}{rl} \mathcal{P}_{z'} & = \{p' \cup q' \mid p' \in \mathcal{P}_{x'}, q' \in \mathcal{P}_{y'}, p' \cap \mathcal{F}_{y'} = q' \cap \mathcal{F}_{x'}\} \\ & = \{(p \cap \mathcal{F}_{x'}) \cup (q \cap \mathcal{F}_{y'}) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_{x'} \cap \mathcal{F}_{y'} = q \cap \mathcal{F}_{y'} \cap \mathcal{F}_{x'}\} \\ & = \{(p \cap \mathcal{F}_{x'}) \cup (q \cap \mathcal{F}_{y'}) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_x \cap \mathcal{F}_y = q \cap \mathcal{F}_y \cap \mathcal{F}_x\} \\ & = \{(p \cap \mathcal{F}_{x'}) \cup (q \cap \mathcal{F}_{y'}) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\} \\ & = \{(p \cup q) \cap (\mathcal{F}_{x'} \cup \mathcal{F}_{y'}) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\} \\ & = \{r \cap \mathcal{F}_{z'} \mid r \in \mathcal{P}_z\} \end{array}$$

□

Proof (of Theorem 8). As $\mathtt{L}' = \mathtt{L}(\bar{\mathtt{L}})$ is defined, for all $1 \leq i \leq n$ we have that $\mathcal{M}_{\mathtt{Z}_i} \preceq \mathcal{M}_{\mathtt{L}_i}$. This implies, with Lemma 1, that $\mathcal{R}(\mathcal{M}_{\mathtt{Z}_i}) \preceq \mathcal{R}(\mathcal{M}_{\mathtt{L}_i})$. Using the hypothesis (ii), we can then apply the Lemma 13 to get that $\mathcal{M}_{Int} \preceq \mathcal{M}_y$ with:

$$\mathcal{M}_y = \mathcal{R}(\mathcal{M}_{\mathtt{L}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{L}_n})$$

$$\mathcal{M}_{Int} = \mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n})$$

We can then remark that:

$$\mathcal{M}_{\mathtt{L}} = \mathcal{M}_{\mathtt{L}}^{Main} \circ_{\mathcal{M}_{\mathtt{L}}^{Glue}} \mathcal{M}_{Int} \qquad \mathcal{M}_{\mathtt{L}'} = \mathcal{M}_{\mathtt{L}}^{Main} \circ_{\mathcal{M}_{\mathtt{L}}^{Glue}} \mathcal{M}_y$$

Using the hypothesis (ii), we can then apply the different theorems in [8] to conclude. □

Appendix F. Proof of Theorem 9

The proof of that compositionality preserves type safety relies on two lemmas, each of them is related to one of the operators used in the composition ($\mathcal{R}$ and $\bullet$).

The following lemma proves that the $\mathcal{R}$ operator preserves type safety.

Lemma 14 (Operation $\mathcal{R}$ on SPLs preserves type safety). *Consider a SPL* $\mathtt{L}$*. Then* $\mathtt{L}$ *being type safe is equivalent to* $\mathcal{R}(\mathtt{L})$ *being type safe.*

Proof. If $\emptyset \in \mathcal{P}_{\mathtt{L}}$, we have $\mathcal{R}(\mathtt{L}) = \mathtt{L}$ which proves the result. Otherwise consider the generator $\mathcal{G}_{\mathcal{R}(\mathtt{L})}$: by construction, we have

$$\mathcal{G}_{\mathcal{R}(\mathtt{L})}(p) = \begin{cases} \emptyset & \text{if } p = \emptyset \\ \mathcal{G}_{\mathtt{L}}(p) & \text{else} \end{cases}$$

Hence, by Definition 5, and because the empty program is well typed, we have the result. □

The following lemma shows an important substitutivity property of the $\bullet$ operator on SPLs. It states that given a type safe SPL, replacing any part of it with a possibly more complex but type safe new part, results in a still type safe SPL.

Lemma 15 (Substitutivity property of $\bullet$ on SPLs). *Suppose given three SPLs* $\mathtt{L}_1$*,* $\mathtt{L}_2$ *and* $\mathtt{L}_3$ *such that: (i)* ***signature***$(\mathtt{L}_2) \preceq$ ***signature***$(\mathtt{L}_3)$*; (ii) the SPLs* $\mathtt{L}_1 \bullet \mathtt{L}_2$ *and* $\mathtt{L}_1 \bullet \mathtt{L}_3$ *are defined; and (iii) the SPLs* $\mathtt{L}_1 \bullet \mathtt{L}_2$ *and* $\mathtt{L}_3$ *are type safe. Then the SPL* $\mathtt{L}_1 \bullet \mathtt{L}_3$ *is type safe.*

Proof. Let write $\mathtt{L} = \mathtt{L}_1 \bullet \mathtt{L}_2$ and $\mathtt{L}' = \mathtt{L}_1 \bullet \mathtt{L}_3$. By construction of the $\bullet$ operator, the declarations and manipulations between $\mathtt{L}_1$ and $\mathtt{L}_2$ (resp. between $\mathtt{L}_1$ and $\mathtt{L}_3$) are distinct. Consequently, we have

$$\mathcal{G}_{\mathtt{L}}(p) = \mathcal{G}_{\mathtt{L}_1}(p \cap \mathcal{F}_{\mathtt{L}_1}) \cup \mathcal{G}_{\mathtt{L}_2}(p \cap \mathcal{F}_{\mathtt{L}_2}) \qquad \text{for all } p \in \mathrm{dom}(\mathcal{G}_{\mathtt{L}})$$
$$\mathcal{G}_{\mathtt{L}'}(p) = \mathcal{G}_{\mathtt{L}_1}(p \cap \mathcal{F}_{\mathtt{L}_1}) \cup \mathcal{G}_{\mathtt{L}_3}(p \cap \mathcal{F}_{\mathtt{L}_3}) \qquad \text{for all } p \in \mathrm{dom}(\mathcal{G}_{\mathtt{L}'})$$

We can then see that $\mathcal{G}_{\mathtt{L}'}$ is total. Indeed, consider $p \in \mathcal{P}_{\mathtt{L}'}$: there exists $p_1 \in \mathcal{P}_{\mathtt{L}_1}$ and $p_3 \in \mathcal{P}_{\mathtt{L}_3}$ with $p = p_1 \cup p_3$ and $p_1 \cap \mathcal{F}_{\mathtt{L}_3} = p_3 \cap \mathcal{F}_{\mathtt{L}_1}$. As ***signature***$(\mathtt{L}_2) \preceq$ ***signature***$(\mathtt{L}_3)$, we have $\mathcal{M}_{\mathtt{L}_2} \preceq \mathcal{M}_{\mathtt{L}_3}$, and so $p_3 \cap \mathcal{F}_{\mathtt{L}_2} \in \mathcal{P}_{\mathtt{L}_2}$. Hence, as $\mathtt{L}$ is type safe, $\mathcal{G}_{\mathtt{L}}$ is total, which means that $\mathcal{G}_{\mathtt{L}}(p_1 \cup (p_3 \cap \mathcal{F}_{\mathtt{L}_2})) =$

$\mathcal{G}_{\mathtt{L}_1}(p_1) \cup \mathcal{G}_{\mathtt{L}_2}(p_3 \cap \mathcal{F}_{\mathtt{L}_2})$ exists. Consequently, $\mathcal{G}_{\mathtt{L}_1}(p_1)$ exists. Moreover, as $\mathtt{L}_3$ is type safe, $\mathcal{G}_{\mathtt{L}_3}$ is also total, which implies that $\mathcal{G}_{\mathtt{L}'}(p)$ exists. Hence, $\mathcal{G}_{\mathtt{L}'}$ is total.

Let finally consider a variant v of $\mathtt{L}'$, corresponding to a product $p \in \mathcal{P}_{\mathtt{L}'}$, and prove that it is well typed. By construction there exists $p_1 \in \mathcal{P}_{\mathtt{L}_1}$ and $p_3 \in \mathcal{P}_{\mathtt{L}_3}$ such that $p = p_1 \cup p_3$ and $p_1 \cap \mathcal{F}_{\mathtt{L}_3} = p_3 \cap \mathcal{F}_{\mathtt{L}_1}$. Consider the variants $v_1 = \mathcal{G}_{\mathtt{L}_1}(p_1)$, $v_2 = \mathcal{G}_{\mathtt{L}_2}(p_3 \cap \mathcal{F}_{\mathtt{L}_2})$ and $v_3 = \mathcal{G}_{\mathtt{L}_3}(p_3)$. We have that $v_1 \cup v_2$ and v_3 are well typed with $\textbf{\textit{signature}}(v_2) \preceq \textbf{\textit{signature}}(v_3)$, while v_3 and v_1 have no common declaration. Consequently, $v = v_1 \cup v_3$ is well typed. $\square$

We can now prove the type safe composition result.

Proof (of Theorem 9). We prove the result by induction on m. The result is clear with $m = 0$, as $\mathtt{L}(\,) = \mathtt{L}$.

Let $m > 0$. By Definition 22, as $\mathtt{L}$ is type safe, then $\mathtt{L}' = \mathtt{L}(\mathtt{Z}_1^\bigstar, \ldots, \mathtt{Z}_n^\bigstar) = \mathcal{N}(\mathtt{L}) \bullet \mathcal{R}(\mathtt{Z}_1^\bigstar) \bullet \cdots \bullet \mathcal{R}(\mathtt{Z}_n^\bigstar) = \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{1 \le i \le n} \mathcal{R}(\mathtt{Z}_i^\bigstar))$ is type safe.

Let $N = \{i \in \mathbb{N} \mid 1 \le i \le n\}$, let $\overline{\mathtt{Z}}^{(m)}$ be the dependencies of $\mathtt{L}_m$ and $I = \{i \in N \mid \mathtt{Z}_i \in \overline{\mathtt{Z}}^{(m)}\}$ and $R = \{i \in N \mid i \notin I \land i \neq m\}$. Clearly R, I, $\{m\}$ form a partition of N; thus, we have

$$\begin{aligned}
\mathtt{L}' = \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{i \in N} \mathcal{R}(\mathtt{Z}_i^\bigstar)) &= \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{i \in R} \mathcal{R}(\mathtt{Z}_i^\bigstar)) \bullet (\mathcal{R}(\mathtt{Z}_m^\bigstar) \bullet (\bigodot_{i \in I} \mathcal{R}(\mathtt{Z}_i^\bigstar))) \\
&= \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{i \in R} \mathcal{R}(\mathtt{Z}_i^\bigstar)) \bullet (\mathcal{R}(\mathtt{Z}_m^\bigstar \bullet (\bigodot_{i \in I} \mathcal{R}(\mathtt{Z}_i^\bigstar)))) \;.
\end{aligned}$$

As $\mathtt{L}_m$ is type safe, by Definition 22, so is the SPL $\mathtt{L}'_m$:

$$\mathtt{L}'_m = \mathcal{R}(\mathcal{N}(\mathtt{L}_m) \bullet (\bigodot_{\mathtt{Z} \in \overline{\mathtt{Z}}^{(m)}} \mathcal{R}(\mathtt{Z}^\bigstar))) \;.$$

It is then easy to see that $\mathcal{R}(\mathtt{Z}_m^\bigstar \bullet (\bigodot_{i \in I} \mathcal{R}(\mathtt{Z}_i^\bigstar))) \preceq \mathtt{L}'_m$. We can apply the Lemma 15 to obtain that the SPL $\mathtt{L}_{\{m\}}$, defined below, is type safe:

$$\begin{aligned}
\mathtt{L}_{\{m\}} &= \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{i \in R} \mathcal{R}(\mathtt{Z}_i^\bigstar)) \bullet (\mathcal{R}(\mathcal{N}(\mathtt{L}_m) \bullet (\bigodot_{\mathtt{Z} \in \overline{\mathtt{Z}}^{(m)}} \mathcal{R}(\mathtt{Z}^\bigstar)))) \\
&= \mathcal{N}(\mathtt{L}) \bullet (\bigodot_{i \in (I \cup R)} \mathcal{R}(\mathtt{Z}_i^\bigstar)) \bullet (\mathcal{R}(\mathcal{N}(\mathtt{L}_m)) \bullet (\bigodot_{\mathtt{Z} \in (\overline{\mathtt{Z}}^{(m)} \setminus \{\mathtt{Z}_i \mid i \in I\})} \mathcal{R}(\mathtt{Z}^\bigstar))) \\
&= \big(\mathcal{N}(\mathtt{L}) \bullet \mathcal{R}(\mathcal{N}(\mathtt{L}_m)) \bullet (\bigodot_{\mathtt{Z} \in (\overline{\mathtt{Z}}^{(m)} \setminus \{\mathtt{Z}_i \mid i \in I\})} \mathcal{R}(\mathtt{Z}^\bigstar)))\big) \bullet (\bigodot_{i \in (I \cup R)} \mathcal{R}(\mathtt{Z}_i^\bigstar)) \;.
\end{aligned}$$

Let $K = \{i \mid \mathtt{Z}'_i \in ((\bigcup_{1\leq i\leq m} \overline{\mathtt{Z}}^{(i)}) \setminus \{\mathtt{Z}_1, \dots, \mathtt{Z}_m\}) \cup \{\mathtt{Z}_m, \dots, \mathtt{Z}_n\}\}$. With the induction hypothesis, we obtain that the SPL defined below is type safe:

$$\begin{aligned}\mathtt{L}_{[1..m]} &= \big(\mathcal{N}(\mathtt{L}) \bullet (\underset{1\leq i\leq m}{\bullet} \mathcal{R}(\mathcal{N}(\mathtt{L}_i)))\big) \bullet (\underset{i\in K}{\bullet} \mathcal{R}(\mathtt{Z}'^{\star}_i)))\big)\\ &= \big(\mathtt{L}(\overline{\mathtt{L}})\big)(\{\mathtt{Z}'_i \in (\bigcup_{1\leq i\leq m} \overline{\mathtt{Z}}^{(i)}) \mid i \in K\})\ . \qquad \square\end{aligned}$$

Appendix G. Proof of Theorem 10

We introduce the proof of the Theorem 10 with the formal definition of the functions used in the formula.

Appendix G.1. Definition of Functions

To simplify our notation, we suppose without loss of generality that all SPLSs have their artifact base only containing deltas with an empty IFJ program signature. The functions used in the $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ constraints are based on simpler getters on the core elements of an SPLS. These core getters are as follows:

$\mathtt{adds}^c(\mathtt{Z}, \mathtt{C})$ returns the set of deltas in $\mathtt{Z}$ that adds the class $\mathtt{C}$.

$\mathtt{adds}^a(\mathtt{Z}, \mathtt{C}, \mathtt{a}, AS)$ returns the set of deltas in $\mathtt{Z}$ that adds the attribute $\mathtt{a}$ to the class $\mathtt{C}$ with the AS declaration.

$\mathtt{removes}^c(\mathtt{Z}, \mathtt{C})$ returns the set of delta in $\mathtt{Z}$ that removes the class $\mathtt{C}$.

$\mathtt{removes}^a(\mathtt{Z}, \mathtt{C}, \mathtt{a})$ returns the set of delta in $\mathtt{Z}$ that removes the class $\mathtt{C}$ or the attribute $\mathtt{a}$ from this class.

$\mathtt{inh}(\mathtt{Z})$ returns inheritance relation graph declared in $\mathtt{Z}$, with each edge annotated with the set of deltas that declare it. We note $\mathtt{path}(\mathtt{Z}, \mathtt{C}_0, \mathtt{C}_n)$ the set of paths $\mathtt{C}_0 \xrightarrow{\overline{\mathtt{d}_1}} \mathtt{C}_1 \xrightarrow{\overline{\mathtt{d}_2}} \dots \xrightarrow{\overline{\mathtt{d}_n}} \mathtt{C}_n$ in the graph $\mathtt{inh}(\mathtt{Z})$.

Products for a specific class. The following equation defines the function $\mathfrak{C}$ which takes two parameters, a SPL signature $\mathtt{Z}$ and a class name $\mathtt{C}$, and that gives the constraint describing the products of $\mathtt{Z}$ whose respective variants contain the class $\mathtt{C}$.

$$\mathfrak{C}(\mathtt{Z}, \mathtt{C}) \triangleq \bigvee_{\mathtt{d}_1} \mathtt{d}_1 \wedge \bigwedge_{\mathtt{d}_2} \neg\mathtt{d}_2 \qquad \text{with} \begin{cases} \mathtt{d}_1 \in \mathtt{adds}^c(\mathtt{Z}, \mathtt{C}), \mathtt{d}_2 \in \mathtt{removes}^c(\mathtt{Z}, \mathtt{C}), \\ \mathtt{d}_1 < \mathtt{d}_2 \end{cases}$$

This formula works as follows. The class $\mathtt{C}$ is present in a variant if and only if it is added by some delta $\mathtt{d}_1$ and never removed by a delta $\mathtt{d}_2$ afterward.

Products for a specific subtyping relation. The following equation defines the function $\mathfrak{I}$ which takes three parameters, a SPL signature $\mathtt{Z}$ and two class names $\mathtt{C}_0$ and $\mathtt{C}_n$, and that gives the constraint describing the products of $\mathtt{Z}$ whose respective variants have $\mathtt{C}$ being a subtype of $\mathtt{C}'$.

$$\mathfrak{I}(\mathtt{Z},\mathtt{C}_0,\mathtt{C}_n) \triangleq \bigvee_{path}\Big(\bigwedge_{1\le i\le n}\big(\bigvee_{\mathtt{d}\in\overline{\mathtt{d}_i}}(\mathtt{d}\wedge\bigwedge_{\mathtt{d}'}\neg\mathtt{d}')\big)\Big)$$

$$\text{with}\begin{cases} path = \mathtt{C}_0 \xrightarrow{\overline{\mathtt{d}_1}} \dots \xrightarrow{\overline{\mathtt{d}_n}} \mathtt{C}_n \in \mathtt{path}(\mathtt{Z},\mathtt{C}_0,\mathtt{C}_n), \\ \mathtt{d}' \in \Big(\bigcup\limits_{\substack{\mathtt{C}'\neq\mathtt{C}_i \\ \mathtt{C}_{i-1}\xrightarrow{\overline{\mathtt{d}}}\mathtt{C}'}} \overline{\mathtt{d}}\Big) \cup \mathtt{removes}^c(\mathtt{Z},\mathtt{C}_{i-1}) \quad , \quad \mathtt{d} < \mathtt{d}' \end{cases}$$

This formula looks at all the possible inheritance path between $\mathtt{C}_0$ and $\mathtt{C}_n$ and checks that each edge $\mathtt{C}_{i-1}{<:}\mathtt{C}_i$ is constructed by some $\mathtt{d}$ and not removed by another delta $\mathtt{d}'$ later, that could change the subtyping relation of $\mathtt{C}_{i-1}$ or entierly remove this class.

Products for a specific lookup result. The following equation defines the function $\mathfrak{L}$ which takes four parameters, a SPL signature $\mathtt{Z}$, a class name $\mathtt{C}_0$, an attribute name $\mathtt{a}$ and an attribute signature AS, and that gives the constraint describing the products of $\mathtt{Z}$ whose respective variants PS are such that $lookup_{PS}(\mathtt{a},\mathtt{C})$ is defined and is equal to AS.

$$\mathfrak{L}^l(\mathtt{Z},\mathtt{C},\mathtt{a},AS) \triangleq \bigvee_{\mathtt{d}_1}\mathtt{d}_1 \wedge \bigwedge_{\mathtt{d}_2}\neg\mathtt{d}_2$$

$$\text{with}\begin{cases} \mathtt{d}_1 \in \mathtt{adds}^a(\mathtt{Z},\mathtt{C},\mathtt{a},AS), \mathtt{d}_2 \in \mathtt{removes}^a(\mathtt{Z},\mathtt{C},\mathtt{a}), \\ \mathtt{d}_1 < \mathtt{d}_2 \end{cases}$$

$$\mathfrak{I}^x(\mathtt{Z},\mathtt{C}_0,\mathtt{C}_n,\mathtt{a}) \triangleq \bigvee_{path}\Big(\bigwedge_{1\le i\le n}\big(\bigvee_{\mathtt{d}\in\overline{\mathtt{d}_i}}(\mathtt{d}\wedge\bigwedge_{\mathtt{d}'}\neg\mathtt{d}')\big)\Big)$$

$$\text{with}\begin{cases} path = \mathtt{C}_0 \xrightarrow{\overline{\mathtt{d}_1}} \dots \xrightarrow{\overline{\mathtt{d}_n}} \mathtt{C}_n \in \mathtt{path}(\mathtt{Z},\mathtt{C}_0,\mathtt{C}_n), \\ \mathtt{d}' \in \Big(\bigcup\limits_{\substack{\mathtt{C}'\neq\mathtt{C}_i \\ \mathtt{C}_{i-1}\xrightarrow{\overline{\mathtt{d}}}\mathtt{C}'}} \overline{\mathtt{d}}\Big) \cup \mathtt{removes}^c(\mathtt{Z},\mathtt{C}_{i-1}) \cup \mathfrak{L}^l(\mathtt{Z},\mathtt{C}_i,\mathtt{a},AS'), \quad \mathtt{d} < \mathtt{d}' \end{cases}$$

$$\mathfrak{L}(\mathtt{Z}, \mathtt{C}, \mathtt{a}, AS) \quad \triangleq \quad \mathfrak{L}^l(\mathtt{Z}, \mathtt{C}, \mathtt{a}, AS) \;\vee\; \bigvee_{\mathtt{C}'} (\mathfrak{I}^x(\mathtt{Z}, \mathtt{C}, \mathtt{C}', \mathtt{a}) \;\wedge\; \mathfrak{L}^l(\mathtt{Z}, \mathtt{C}', \mathtt{a}, AS))$$

The definition of $\mathfrak{L}$ uses two annex functions. The first one $\mathfrak{L}^l$ takes in input an SPLS $\mathtt{Z}$, a class name $\mathtt{C}$, an attribute name $\mathtt{a}$ and an attribute signature AS, and states for which products of $\mathtt{Z}$, the corresponding variant contains $\mathtt{C}$ with $\mathtt{a}$ declared as AS. The definition of this annex function is constructed similarly to the definition of the $\mathfrak{C}$ function. The second annex function is $\mathfrak{I}^x$ is an extension of the $\mathfrak{I}$ function where we ensure that the attribute $\mathtt{a}$ is not redefined after $\mathtt{C}_0$ in the inheritance path. Then, the definition of $\mathfrak{L}(\mathtt{Z}, \mathtt{C}, \mathtt{a}, AS)$ simply checks if the attribute $\mathtt{a}$ is declared locally to $\mathtt{C}$ (using the $\mathfrak{L}^l$ function), or if it is declared in a superclass $\mathtt{C}'$ of $\mathtt{C}$ and not redefined in between in the inheritance graph (using the $\mathfrak{I}^x$ function).

Appendix G.2. Proof of the Theorem 10

We first recall the statement of this theorem.

Suppose given two SPL signatures $\mathtt{Z}_1$ *and* $\mathtt{Z}_2$ *with* $\mathcal{G}_{\mathtt{Z}_1}$ *and* $\mathcal{G}_{\mathtt{Z}_2}$ *total. Then the two following statements are equivalent:*

(i) $\mathtt{Z}_1 \preceq \mathtt{Z}_2$

(ii) $\mathcal{F}_{\mathtt{Z}_1} \subseteq \mathcal{F}_{\mathtt{Z}_2}$, $(\exists_{f \in \mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1}} f.\mathfrak{P}(\mathtt{Z}_2)) \Leftrightarrow \mathfrak{P}(\mathtt{Z}_1)$ *and* $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ *are valid.*

To improve the proof readability, we write $\mathfrak{E}$ the formula

$$(\exists_{f \in \mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1}} f.\mathfrak{P}(\mathtt{Z}_2)) \Leftrightarrow \mathfrak{P}(\mathtt{Z}_1)$$

Proof. In case the SPLS $\mathtt{Z}_2$ has no products, the result is straightforward. Let now assume that $\mathtt{Z}_2$ has at least one product: we prove the equivalence of the two properties by proving each implication independently.

$\Rightarrow$. By hypothesis, we have $\mathtt{Z}_1 \preceq \mathtt{Z}_2$, which implies that $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$ (Definition 10). Hence, by Definition 7, we have $\mathcal{F}_{\mathtt{Z}_1} \subseteq \mathcal{F}_{\mathtt{Z}_2}$. Moreover, consider the following reductio ad absurdum: let consider a solution $\sigma \subseteq \mathcal{F}_{\mathtt{Z}_1}$ of $\neg(\mathfrak{E})$. We have two cases:

(i) either $\sigma \notin \mathcal{P}_{\mathtt{Z}_1}$ and there exists $\sigma' \subseteq \mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1}$ such that $\sigma \cup \sigma' \in \mathcal{P}_{\mathtt{Z}_2}$. By definition of $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$, we have

$$\sigma \cup \sigma' \in \mathcal{P}_{\mathtt{Z}_2} \quad \Rightarrow \quad \sigma = (\sigma \cup \sigma') \cap \mathcal{F}_{\mathtt{Z}_1} \in \mathcal{P}_{\mathtt{Z}_1}$$

(ii) either $\sigma \in \mathcal{P}_{\mathtt{Z}_1}$ and for all $\sigma' \subseteq \mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1}$, we have $\sigma \cup \sigma' \notin \mathcal{P}_{\mathtt{Z}_2}$. However, by definition of $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$, there exists $\sigma'' \in \mathcal{P}_{\mathtt{Z}_2}$ with $\sigma'' \cap \mathcal{F}_{\mathtt{Z}_1} = \sigma$: using $\sigma' = \sigma'' \cap (\mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1})$, we have a contradiction.

Hence, $\mathfrak{E}$ is valid.

We now prove that $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ is valid. Let consider a solution σ of $\mathfrak{P}(\mathtt{Z}_2)$ and the product $p = \sigma \cap \mathcal{F}_{\mathtt{Z}_2}$. Let write $PS_1 = \mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1})$ and $PS_2 = \mathcal{G}_{\mathtt{Z}_2}(p)$. By Definition 10, we have that $PS_1 \preceq PS_2$. Hence, by Definition 9, we have:

(i) $\mathrm{dom}(PS_1) \subseteq \mathrm{dom}(PS_2)$, which means that the following statement holds:

$$\sigma \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} (\mathfrak{C}(\mathtt{Z}_1, \mathtt{C}) \Rightarrow \mathfrak{C}(\mathtt{Z}_2, \mathtt{C}))$$

(ii) $<:_{PS_1} \subseteq <:_{PS_2}$, which means that the following statement holds:

$$\sigma \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} \bigwedge_{\mathtt{C}' \in \mathtt{gclass}(\mathtt{Z}_1)} (\mathfrak{I}(\mathtt{Z}_1, \mathtt{C}, \mathtt{C}') \Rightarrow \mathfrak{I}(\mathtt{Z}_2, \mathtt{C}, \mathtt{C}'))$$

(iii) for all class name $\mathtt{C} \in \mathrm{dom}(PS_2)$ and all attribute name $\mathtt{a} \in \mathrm{dom}(\mathtt{C}_{PS_2})$, we have that $\mathit{lookup}_{PS_1}(\mathtt{a}, \mathtt{C})$ is defined and $\mathit{lookup}_{PS_2}(\mathtt{a}, \mathtt{C}) = \mathit{lookup}_{PS_1}(\mathtt{a}, \mathtt{C})$. This means that the following statement holds:

$$\sigma \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} \bigwedge_{(\mathtt{a}, AS) \in \mathtt{gatt}(\mathtt{Z}_1, \mathtt{C})} (\mathfrak{L}(\mathtt{Z}_1, \mathtt{C}, \mathtt{a}, AS) \Rightarrow \mathfrak{L}(\mathtt{Z}_2, \mathtt{C}, \mathtt{a}, AS))$$

Consequently, the formula $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ is valid.

$\Leftarrow$. Let first note that $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$ is a direct consequence of $\mathcal{F}_{\mathtt{Z}_1} \subseteq \mathcal{F}_{\mathtt{Z}_2}$ and $\mathfrak{E}$. Let first consider a product $p \in \mathcal{P}_{\mathtt{Z}_1}$. As $\mathfrak{E}$ is valid, there exists $\sigma' \subseteq (\mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1})$ such that $p \cup \sigma' \in \mathcal{P}_{\mathtt{Z}_2}$, and so $\mathcal{P}_{\mathtt{Z}_1} \subseteq \{p \cap \mathcal{F}_{\mathtt{Z}_1} \mid p \in \mathcal{P}_{\mathtt{Z}_2}\}$. Let then consider a product $p \in \mathcal{P}_{\mathtt{Z}_2}$. By construction, $p' = p \setminus (\mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1})$ is a solution of $\exists_{f \in \mathcal{F}_{\mathtt{Z}_2} \setminus \mathcal{F}_{\mathtt{Z}_1}} f.\mathfrak{P}(\mathtt{Z}_2)$, and, as $\mathfrak{E}$ is valid, p' is a product of $\mathcal{P}_{\mathtt{Z}_1}$. Consequently, we have $\mathcal{P}_{\mathtt{Z}_1} \supseteq \{p \cap \mathcal{F}_{\mathtt{Z}_1} \mid p \in \mathcal{P}_{\mathtt{Z}_2}\}$, which gives us $\mathcal{M}_{\mathtt{Z}_1} \preceq \mathcal{M}_{\mathtt{Z}_2}$.

Let now consider a product p of $\mathtt{Z}_2$ and prove that $\mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1}) \preceq \mathcal{G}_{\mathtt{Z}_2}(p)$. We write $PS_1 = \mathcal{G}_{\mathtt{Z}_1}(p \cap \mathcal{F}_{\mathtt{Z}_1})$ and $PS_2 = \mathcal{G}_{\mathtt{Z}_2}(p)$. By Definition 10, we have three cases to prove:

(i) as $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ is valid we have:

$$p \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} (\mathfrak{C}(\mathtt{Z}_1, \mathtt{C}) \Rightarrow \mathfrak{C}(\mathtt{Z}_2, \mathtt{C}))$$

This directly implies that $\mathrm{dom}(PS_1) \subseteq \mathrm{dom}(PS_2)$.

(ii) as $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ is valid we have:

$$p \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} \bigwedge_{\mathtt{C}' \in \mathtt{gclass}(\mathtt{Z}_1)} (\mathfrak{I}(\mathtt{Z}_1, \mathtt{C}, \mathtt{C}') \Rightarrow \mathfrak{I}(\mathtt{Z}_2, \mathtt{C}, \mathtt{C}'))$$

This implies that if $\mathtt{C} <:_{PS_1} \mathtt{C}'$, then $\mathtt{C} <:_{PS_2} \mathtt{C}'$, which means that $<:_{PS_1} \subseteq <:_{PS_2}$.

(iii) as $\mathfrak{S}(\mathtt{Z}_1, \mathtt{Z}_2)$ is valid we have:

$$p \vdash \bigwedge_{\mathtt{C} \in \mathtt{gclass}(\mathtt{Z}_1)} \bigwedge_{(\mathtt{a}, AS) \in \mathtt{gatt}(\mathtt{Z}_1, \mathtt{C})} (\mathfrak{L}(\mathtt{Z}_1, \mathtt{C}, \mathtt{a}, AS) \Rightarrow \mathfrak{L}(\mathtt{Z}_2, \mathtt{C}, \mathtt{a}, AS))$$

This means that if $\mathit{lookup}_{PS_1}(\mathtt{a}, \mathtt{C})$ is defined and is equal to AS, then $\mathit{lookup}_{PS_2}(\mathtt{a}, \mathtt{C})$ returns the same value.

$\square$

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices & Patterns, Addison Wesley Longman, 2001.

[2] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering - Foundations, Principles, and Techniques, Springer, 2005.

[3] S. Apel, D. S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Springer, 2013.

[4] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-Oriented Programming of Software Product Lines, in: Software Product Lines: Going Beyond (SPLC 2010), Vol. 6287 of Lecture Notes in Computer Science, 2010, pp. 77–91. doi:10.1007/978-3-642-15579-6_6.

[5] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (2004) 355–371. doi:10.1109/TSE.2004.23.

[6] G. Holl, P. Grünbacher, R. Rabiser, A systematic review and an expert survey on capabilities supporting multi product lines, Information & Software Technology 54 (8) (2012) 828–852. doi:10.1016/j.infsof.2012.02.002.

[7] F. Damiani, I. Schaefer, T. Winkelmann, Delta-oriented multi software product lines, in: Proceedings of the 18th International Software Product Line Conference - Volume 1, SPLC '14, ACM, 2014, pp. 232–236. doi:10.1145/2648511.2648536.

[8] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, G. Saake, Feature-model interfaces: The highway to compositional analyses of highly-configurable systems, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, 2016, pp. 667–678. doi:10.1145/2884781.2884823.

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

[10] F. Damiani, M. Lienhardt, L. Paolini, A formal model for multi spls, in: 7th International Conference on Fundamentals of Software Engineering (FSEN), Vol. 10522 of Lecture Notes in Computer Science, Springer, Berlin, Germany, 2017, pp. 67–83. doi:10.1007/978-3-319-68972-2_5.

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

[12] M. Lienhardt, D. Clarke, 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, Vol. 7609 of Lecture Notes in Computer Science, 2012, pp. 178–192. doi:10.1007/978-3-642-34026-0_14.

[13] D. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of International Software Product Line Conference (SPLC), Vol. 3714 of Lecture Notes in Computer Science, Springer, 2005, pp. 7–20. doi:10.1007/11554844_3.

[14] M. Rosenmüller, N. Siegmund, S. S. ur Rahman, C. Kästner, Modeling dependent software product lines, in: Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for

Product Line Engineering (McGPLE), MIP-0802, Department of Informatics and Mathematics, University of Passau, 2008, pp. 13–18.

[15] F. Damiani, A. Poetzsch-Heffter, Y. Welsch, A type system for checking specialization of packages in object-oriented programming, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, ACM, New York, NY, USA, 2012, pp. 1737–1742. doi:10.1145/2245276.2232058.

[16] F. Damiani, I. Schaefer, 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, Vol. 7609 of Lecture Notes in Computer Science, Springer, 2012, pp. 193–207. doi:10.1007/978-3-642-34026-0_15.

[17] F. Damiani, M. Lienhardt, On type checking delta-oriented product lines, in: Integrated Formal Methods: 12th International Conference, iFM 2016, Vol. 9681 of Lecture Notes in Computer Science, Springer, 2016, pp. 47–62. doi:10.1007/978-3-319-33693-0_4.

[18] B. Delaware, W. R. Cook, D. Batory, Fitting the pieces together: A machine-checked model of safe composition, in: ESEC/FSE, ACM, 2009, pp. 243–252. doi:10.1145/1595696.1595733.

[19] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, GPCE '07, ACM, 2007, pp. 95–104. doi:10.1145/1289971.1289989.

[20] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6:1–6:45. doi:10.1145/2580950.

[21] R. Schröter, T. Thüm, N. Siegmund, G. Saake, Automated analysis of dependent feature models, in: The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, ACM, 2013, pp. 9:1–9:5. doi:10.1145/2430502.2430515.

[22] R. Schröter, N. Siegmund, T. Thüm, Towards modular analysis of multi product lines, in: Proceedings of the 17th International Software Product Line Conference Co-located Workshops, SPLC'13, ACM, 2013, pp. 96–99. doi:10.1145/2499777.2500719.

[23] R. Schröter, N. Siegmund, T. Thüm, G. Saake, Feature-context interfaces: Tailored programming interfaces for spls, in: Proceedings of the 18th International Software Product Line Conference - Volume 1, SPLC'14, ACM, 2014, pp. 102–111. doi:10.1145/2648511.2648522.

[24] I. Schaefer, F. Damiani, Pure delta-oriented programming, in: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD '10, ACM, 2010, pp. 49–56. doi:10.1145/1868688.1868696.

[25] C. Kästner, K. Ostermann, S. Erdweg, A variability-aware module system, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, 2012, pp. 773–792. doi:10.1145/2384616.2384673.

[26] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela, Software diversity: state of the art and perspectives, International Journal on Software Tools for Technology Transfer 14 (5) (2012) 477–495. doi:10.1007/s10009-012-0253-y.

[27] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, F. Damiani, 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, ACM, 2014, pp. 63–74. doi:10.1145/2647508.2647512.

[28] T. Winkelmann, J. Koscielny, C. Seidl, S. Schuster, F. Damiani, I. Schaefer, Parametric deltaj 1.5: Propagating feature attributes into implementation artifacts, in: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Vol. 1559 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 40–54.
URL `http://ceur-ws.org/Vol-1559/paper04.pdf`

[29] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: Formal Methods for Components and Objects, Vol. 6957 of Lecture Notes in Computer Science, Springer, 2012, pp. 142–164. doi:10.1007/978-3-642-25271-6_8.

[30] F. Damiani, M. Lienhardt, R. Muschevici, I. Schaefer, An extension of the ABS toolchain with a mechanism for type checking spls, in: Integrated Formal Methods - 13th International Conference, IFM 2017,

Proceedings, Vol. 10510 of Lecture Notes in Computer Science, Springer, 2017, pp. 111–126. doi:10.1007/978-3-319-66845-1_8.

[31] F. Damiani, D. Faitelson, C. Gladisch, S. Tyszberowicz, A novel model-based testing approach for software product lines, Software & Systems Modeling 16 (4) (2017) 1223–1251. doi:10.1007/s10270-016-0516-2.

[32] D. Faitelson, S. S. Tyszberowicz, Data refinement based testing, Int. J. Systems Assurance Engineering and Management 2 (2) (2011) 144–154. doi:10.1007/s13198-011-0060-y.

[33] W. P. de Roever, K. Engelhardt, Data Refinement: Model-oriented Proof Theories and their Comparison, Vol. 46 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[34] E. Kamburjan, R. Hähnle, Uniform modeling of railway operations, in: Proc. of FTSCS 2016, Vol. 694 of CCIS, Springer, 2017, pp. 55–71. doi:10.1007/978-3-319-53946-1_4.

[35] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, Interoperability of software product line variants, in: Proceeedings of the 22nd International Conference on Systems and Software Product Line - Volume 1, SPLC 2018, ACM, 2018, pp. 264–268. doi:10.1145/3233027.3236401.

[36] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, Same Same But Different: Interoperability of Software Product Line Variants, Springer International Publishing, Cham, 2018, pp. 99–117. doi:10.1007/978-3-319-98047-8_7.

[37] M. Lienhardt, F. Damiani, S. Donetti, L. Paolini, Multi software product lines in the wild, in: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, ACM, 2018, pp. 89–96. doi:10.1145/3168365.3170425.

[38] C. Chesta, F. Damiani, L. Dobriakova, M. Guernieri, S. Martini, M. Nieke, V. Rodrigues, S. Schuster, A toolchain for delta-oriented modeling of software product lines, in: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Proceedings, Part II, Vol. 9953 of Lecture Notes in Computer Science, Springer, 2016, pp. 497–511. doi:10.1007/978-3-319-47169-3_40.

[39] M. Lienhardt, F. Damiani, L. Testa, G. Turin, On checking delta-oriented product lines of statecharts, Science of Computer Programming 166 (2018) 3 – 34. doi:10.1016/j.scico.2018.05.007.

[40] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231 – 274. doi:10.1016/0167-6423(87)90035-9.

[41] YAKINDU Statechart Tools, https://www.itemis.com/en/yakindu/state-machine/ (2017).

[42] Gentoo Foundation, https://wiki.gentoo.org/wiki/Portage (2017).

[43] Gentoo Foundation, https://gentoo.org (2017).

[44] T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, S. Krüger, Variability hiding in contracts for dependent spls, in: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS'16, ACM, 2016, pp. 97–104. doi:10.1145/2866614.2866628.

[45] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, S. Apel, Variability encoding: From compile-time to load-time variability, Journal of Logical and Algebraic Methods in Programming 85 (1, Part 2) (2016) 125–145. doi:10.1016/j.jlamp.2015.06.007.

[46] R. Hähnle, I. Schaefer, A Liskov Principle for Delta-Oriented Programming, in: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I, Vol. 7609 of Lecture Notes in Computer Science, Springer, 2012, pp. 32–46. doi:10.1007/978-3-642-34026-0_4.

[47] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, I. C. Yu, A transformational proof system for delta-oriented programming, in: Proc. of SPLC 2012 - Volume 2, ACM, 2012, pp. 53–60. doi:10.1145/2364412.2364422.

[48] R. Hähnle, I. Schaefer, R. Bubel, Reuse in software verification by abstract method calls, in: Proceedings of International Conference on Automated Deduction, CADE'13, Vol. 7898 of Lecture Notes in Computer Science, Springer, 2013, pp. 300–314. doi:10.1007/978-3-642-38574-2_21.

[49] R. Bubel, F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, I. C. Yu, Proof repositories for compositional verification of evolving software systems - managing change when proving software correct, Transactions on Foundations for Mastering Change I 1 (2016) 130–156. doi:10.1007/978-3-319-46508-1_8.

[50] C. C. Din, E. B. Johnsen, O. Owe, I. C. Yu, A modular reasoning system using uninterpreted predicates for code reuse, Journal of Logical and Algebraic Methods in Programming 95 (2018) 82 – 102. doi:10.1016/j.jlamp.2017.11.004.