A Formal Model for Multi SPLs

Ferruccio Damiani, Michael Lienhardt, Luca Paolini

A Formal Model for Multi SPLs[⋆]

Ferruccio Damiani, Michael Lienhardt, and Luca Paolini

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

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.

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 [7,19,2]. An SPL consists of: (i) a *feature model* defining the set of variants in terms of *features* (each feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*); (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) [21], [2, Sect. 6.6.1] is a flexible and 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 modifications to a program (e.g., for Java programs, a delta can add, remove or modify classes and interfaces). The configuration knowledge of a delta-oriented SPL defines the generator by associating to each delta an *activation condition* over the features (i.e., a set of products) and specifying an *application ordering* between deltas. DOP supports the automatic generation of variants based on a selection of features: once

[⋆] 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); and Ateneo/CSP D16D15000360005 project RunVar (runvar-project.di.unito.it).

a user selects a product, the corresponding variant is derived by applying the deltas with a satisfied activation condition to the base program according to the application ordering. Moreover, DOP is a generalization of *Feature-Oriented Programming* (FOP) [4], [2, 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 [13]. 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. 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. [24] on compositional analysis of feature models, and on the delta-oriented programming core calculus IFΔJ by Bettini et al. [5], 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.

Section 2 provides some background. Section 3 formalizes the main concepts proposed in the paper by introducing the Imperative Featherweight Multi Delta Java (IFMΔJ) calculus, which extends IFΔJ to implement MPLs. Section 4 illustrates how the concepts of SPL signature, dependent SPL, and SPL composition support compositionality of existing SPL analysis, like feature model analysis or type checking. Section 5 discusses related work.

2 Background and Running Example

2.1 IFΔJ: a Formal Foundation for Delta-Oriented SPLs

IFΔJ [5] is a core calculus for delta-oriented SPLs where variants are written in IFJ (an imperative version of FJ [14]). The abstract syntax of IFJ is given in Figure 1 (explanations are given in the caption)—following [14], we use the overline notation for (possibly empty) sequences of elements: for instance $\overline{e}$ stands for a sequence of expressions. The empty sequence is denoted by $\emptyset$. Type system, operational semantics, and type soundness for IFJ are given in [5].

The abstract syntax of IFΔJ SPLs is given in Figure 2 (explanations are given in the caption). 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).

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

Fig. 1. Syntax of IFJ. A program P is a sequence of class declarations $\overline{CD}$. A class declaration comprises the name `C` of the class, the name of the superclass (which must always be specified, even if it is the built-in class `Object`), and a list of attribute (field or method) declarations $\overline{AD}$. Variables `x` include the special variable `this` (implicitly bound in any method declaration MD), which may not be used as the name of a method's formal parameter. All fields and methods are public, there is no field shadowing, there is no method overloading, and each class is assumed to have an implicit constructor that initialized all fields to **null**. The subtyping relation $<:$ on classes, which is the reflexive and transitive closure of the immediate subclass relation (given by the **extends** clauses in class declarations), is supposed to be acyclic.

$$\begin{array}{lll}
LD & ::= \textsf{line}\ \texttt{L}\ \{\mathcal{M}\ \mathcal{K}\ AB\} & \text{SPL Delaration} \\
AB & ::= P\ \overline{DD} & \text{Artifact Base} \\
DD & ::= \textsf{delta}\ \texttt{d}\ \{\ \overline{CO}\ \} & \text{Delta Declaration} \\
CO & ::= \textsf{adds}\ CD \mid \textsf{removes}\ \texttt{C} \mid \textsf{modifies}\ \texttt{C}\ [\textsf{extends}\ \texttt{C}']\ \{\ \overline{AO}\ \} & \text{Class Operation} \\
AO & ::= \textsf{adds}\ AD \mid \textsf{removes}\ \texttt{a} \mid \textsf{modifies}\ MD & \text{Attribute Operation}
\end{array}$$

Fig. 2. Syntax of IFΔJ SPLs. An SPL declaration comprises the name `L` of the product line, a feature model $\mathcal{M}$, 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 name* `a` is either a field name `f` or a method name `m`. An attribute operation can add or remove fields and methods, and modify the implementation of a method by replacing its body. The new body may call the special method `original`, which is implicitly bound to the previous implementation of the method and may not be used as the name of a method.

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 declaration the set of products that activate it (the activation condition), and* $<_x$ *is an ordering between deltas (the application ordering).*

These representations 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 with cross tree-constraints) or as propositional formulas Φ where variables are feature names f (see, e.g., [3] for a discussion on other possible representations):

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

To avoid over-specification, the ordering $<_x$ may be partial. We assume *unambiguity* of the SPL, i.e., for each product, any total ordering of the activated deltas that respects $<_x$ generates the same variant (see [18,5] for effective means to ensure unambiguity). In 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.

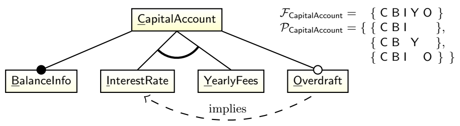
Feature model, configuration knowledge and artifact base of an SPL named $\mathtt{L}$ are denoted by $\mathcal{M}_{\mathtt{L}} = (\mathcal{F}_{\mathtt{L}}, \mathcal{P}_{\mathtt{L}})$, $\mathcal{K}_{\mathtt{L}} = (\alpha_{\mathtt{L}}, <_{\mathtt{L}})$ and $AB_{\mathtt{L}}$, respectively. In order to define the generator $\mathcal{G}_{\mathtt{L}}$ of an SPL $\mathtt{L}$, we first introduce the auxiliary notions of delta applicability and delta application. A delta $\mathtt{d}$ is *applicable* to a program P iff each class to be added does not exist; each class to be removed or modified exists; and (for every class-modify operation): each method or field to be added does not exist; each method or field to be removed exists; each method to be modified exists and has the same header specified in the method-modify operation. If $\mathtt{d}$ is applicable to P, then the *application* of $\mathtt{d}$ to P is the program, denoted by $\mathtt{d}(P)$, obtained from P by applying all the operations in $\mathtt{d}$—otherwise $\mathtt{d}(P)$ is undefined.

Definition 3 (Generator of an SPL [5]). *The* generator *of* $\mathtt{L}$*, denoted by* $\mathcal{G}_{\mathtt{L}}$*, is the mapping that associates each product* p *of* $\mathtt{L}$ *to the IFJ program* $\mathtt{d}_n(\cdots \mathtt{d}_1(P)\cdots)$*, where* P *is the base program of* $\mathtt{L}$ *and* $\mathtt{d}_1 \ldots, \mathtt{d}_n$ $(n \geq 0)$ *are the deltas of* $\mathtt{L}$ *activated by* p*, listed according to the application order.*

The generator $\mathcal{G}_{\mathtt{L}}$ may be partial since, for some product of $\mathtt{L}$, a delta DD_i $(1 \leq i \leq n)$ may not be applicable to the intermediate variant $DD_{i-1}(\cdots\ DD_1(P)\cdots)$ thus making $\mathcal{G}_{\mathtt{L}}$ undefined for that product.

The running example of this paper is based on bank accounts. Figure 3 illustrates an SPL of capital accounts (CapitalAccount, on the left) and an SPL of financial accounts (FinancialAccount, on the right)—explanations are given in the caption. To make the example more readable, in the artifact bases we use Java syntax for field initialization, primitive data types, strings and sequential composition—encoding in IFΔJ syntax is straightforward (see [5]).

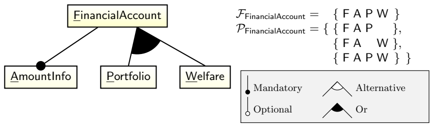
Remark 1 (Base program and empty product). In order to simplify the presentation, the formal definitions in the rest of this document assume that: (i) the base program is always the empty program; (ii) no delta $\mathtt{d}$ is activated by the empty product (i.e., $\emptyset \notin \alpha_{\mathtt{L}}(\mathtt{d})$ for all $\mathtt{d}$); and (iii) $\mathcal{G}_{\mathtt{L}}(\emptyset) = \emptyset$, even when $\emptyset$ is not a product. Note that these assumptions are not restrictive. In particular, the base program of any SPL $\mathtt{L}$ can be always encoded as an extra delta (the *base* delta) with distinguished name $\mathtt{d}_{\mathtt{L}}$ such that $\alpha_{\mathtt{L}}(\mathtt{d}_{\mathtt{L}}) = \mathcal{P}_{\mathtt{L}}$ and $\mathtt{d}_{\mathtt{L}}$ is the minimum according to $<_{\mathtt{L}}$.



$$\mathcal{F}_{\mathsf{CapitalAccount}} = \{\,\mathsf{C\ B\ I\ Y\ O}\,\}$$
$$\mathcal{P}_{\mathsf{CapitalAccount}} = \{\,\{\,\mathsf{C\ B\ I}\,\},\ \{\,\mathsf{C\ B\ Y}\,\},\ \{\,\mathsf{C\ B\ I\ O}\,\}\,\}$$

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

```
class CapAccount extends Object { // Base Program
  String identity; double balance=0.0;
  MyDate lastUpdate=MyDate.today();
  void withdraw(double x){if(x>0) balance−=x;}
}                                        // Deltas
delta dInterest {
  modifies class CapAccount{
   adds double yearRate=0.05; adds double opFees=1;
   adds void interestUpdate(double rate){
     double range=lastUpdate.daysSince()/365;
     lastUpdate=MyDate.today();
     balance+= balance∗rate∗range;}
   modifies void withdraw(double x){
     interestUpdate(yearRate); original(x+opFees); }}}
delta dFixFees {
  modifies class CapAccount {
   adds double yearFees=10.0;
   adds MyDate yearPaid=MyDate.currentYear();
   adds void withdraw(double x){
    balance−=yearFees∗(yearPaid.yearsSince());
    yearPaid=MyDate.currentYear(); original(x);}}}
 delta dOverdraft{
   modifies class CapAccount { adds double maxOver=100.0, negativeRate=0.10;
    adds void negUpdate(){ if(balance<0){interestUpdate(−negativeRate);}}
    modifies void withdraw(double x){ negUpdate(); if(x<balance+maxOver−opFees) original(x);}}}
```



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

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

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

Fig. 3. Left: CapitalAccount SPL: feature model $\mathcal{M}_{\mathsf{CapitalAccount}}$ (top), configuration knowledge $\mathcal{K}_{\mathsf{CapitalAccount}}$ (middle), and artifact base $AB_{\mathsf{CapitalAccount}}$ (bottom). This SPL 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). 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 feaure Overdraft, which allows to withdraw more money than that available, requires feaure InterestRate in order to apply a negative interest. **Right:** FinancialAccount SPL: $\mathcal{M}_{\mathsf{FinancialAccount}}$ (top), $\mathcal{K}_{\mathsf{FinancialAccount}}$ (middle), and $AB_{\mathsf{FinancialAccount}}$ (bottom). This SPL provides a class FinAccount for investment product managing bank accounts. The mandatory feature AmountInfo provides basic fields (`identity`, `liquidity`). It must be flanked by at least one feature between Portfolio and Welfare. The latter provides a list of welfare products. The former provides a list of financial products.

2.2 Feature Model Composition and Feature Model Interfaces

Recently, Schröter et al. [24] considered a notion of feature model composition through aggregation (i.e., by inclusion of one feature model into another feature model [20]) and proposed to use it in combination with a notion of feature model interface in order to support compositional analyses of feature models.

Definition 4 (Feature model composition [24]). *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* glueproviso $\mathcal{F}_{Glue} \subseteq \mathcal{F}_x \cup \mathcal{F}_y$. *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 *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 (thus $\mathcal{P}_y$ core features are not necessarily core in the composed feature model). Operation $\bullet$ is similar to a cross product from relational algebra and creates all combinations between both product sets.

The feature model $\mathcal{M}_{Glue}$ describes a parent-child relationship and other constraints between $\mathcal{M}_x$ and $\mathcal{M}_y$ in order to connect them.

Definition 5 (Feature model interface [24]). *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* $\mathcal{P}_{Int} = \{p \cap \mathcal{F}_{Int} \mid p \in \mathcal{P}_x\}$*.*

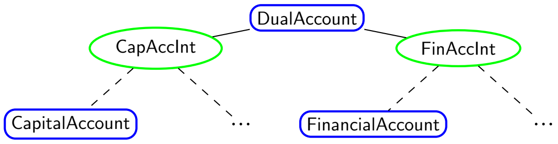
Remark 2 (Feature disjointness). As pointed out in [24, Sect. 4.1, second to last paragraph] the compositional results about $\circ$ "are based on the assumption that $\mathcal{F}_x$ and $\mathcal{F}_y$ do not share features (i.e. $\mathcal{F}_x \cap \mathcal{F}_y = \emptyset$)". In the rest of this document, the use of $\circ$ always relies on this *feature disjointedness* assumption.

3 IFMΔJ: a Core Calculus for MPLs

The example presented in Figure 3 introduces two SPLs, CapitalAccount and FinancialAccount, describing two kinds of bank accounts: it would make perfect sense to combine these two SPLs in order to obtain an SPL describing a bank account with functionalities described in both SPLs.

In a first approach, one could define a new SPL DualAccount that uses (i.e., depends on) the two bank account SPLs presented in Figure 3 to define a new class that implements the different features defined in the two SPLs. We call an SPL with such dependencies a *Dependent SPL*.However, such an approach is not satisfactory as it couples too strongly DualAccount to its SPLs: DualAccount is set to use the CapitalAccount and FinancialAccount SPLs and cannot change even if a more efficient implementation of these SPLs comes up. To deal with this issue, we introduce the notion of *SPL signature* which is used to specify the APIs on which a Dependent SPL depends; then any SPL that implements such signature can fulfill the dependencies of a Dependent SPL.

Hence, our approach to define the DualAccount Dependent SPL follows the structure presented on the right: DualAccount depends on two



$PS ::= \overline{CS}$	Program Signature
$CS ::= \textsf{\textbf{class}}\ \mathtt{C}\ \textsf{\textbf{extends}}\ \mathtt{C}\ \{\ \overline{AS}\ \}$	Class Signature
$AS ::= FD \mid MH$	Attribute (Field or Method) Signature
$LS\ ::= \textsf{\textbf{sig}}\ \mathtt{Z}\ \{\mathcal{M}\ \mathcal{K}\ ABS\}$	SPL Signature Declaration
$ABS ::= PS\ \overline{DS}$	AB Signature
$DS\ ::= \textsf{\textbf{delta}}\ \mathtt{d}\ \{\ \overline{COS}\ \}$	Delta Signature
$COS ::= \textsf{\textbf{adds}}\ CS \mid \textsf{\textbf{removes}}\ \mathtt{C} \mid \textsf{\textbf{modifies}}\ \mathtt{C}\ [\textsf{\textbf{extends}}\ \mathtt{C}']\ \{\ \overline{AOS}\ \}$	CO Signature
$AOS ::= \textsf{\textbf{adds}}\ AS \mid \textsf{\textbf{removes}}\ \mathtt{a}$	AO Signature
$LD\ ::= \textsf{\textbf{line}}\ \mathtt{L}\ (\overline{\mathtt{Z}})\ \{\mathcal{M}_{Main}\ \mathcal{M}_{Glue}\ \mathcal{K}\ AB\}$	Dependent SPL Delaration

Fig. 4. Syntax of IFMΔJ. Program signatures (top). SPL signature declarations (middle). Dependent SPL declarations (bottom)—the extensions with respect to IFΔJ SPLs (given in Figure 2, with the syntax of artifact bases *AB*) are highlighted in grey.

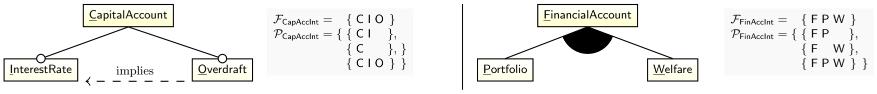
SPL signatures: CapAccInt specifies the API requested by DualAccount for the capital account backend implementation, while FinAccInt specifies the API requested by DualAccount for the financial account implementation. Then these two signatures are implemented by CapitalAccount and FinancialAccount respectively,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

We structure the presentation of our model as follows: first we introduce the concept of SPL signature (SPLS) and formally define when an SPL implements an SPL signature; second we define the notion of Dependent SPL (DPL) as we just presented; and finally, we demonstrate how to generate the variants of a DPL.

3.1 SPL Signatures

An *SPL signature* (SPLS) describes the API of an SPL and is structured like an SPL with a feature model, configuration knowledge, and an artifact base. Its difference with an SPL lies in the fact that its artifact base does not include the implementation of methods. Figure 4 (middle) gives the abstract syntax of SPLSs which uses *program signatures*, presented in Figure 4 (top), to construct their artifact bases. A program signature is a program deprived of method bodies. An SPLS declaration LS comprises the name Z of the SPLS, a feature model $\mathcal{M}$, configuration knowledge $\mathcal{K}$ and an *artifact base signature* ABS which, in turn, comprises a program signature PS and a set of delta signatures $\overline{DS}$—a *delta signature* DS is a delta deprived of method-modifies operations and method bodies.

An SPL L implements an SPLS Z when all the declarations in Z are implemented in L. I.e., when all the products of Z can be extended in a product of L and for each variant of Z, all of its declared elements are implemented in the corresponding variant of L. We first define the *generator* of an SPLS (in order to define what are its variants and their declaration), and then present the definition of the *interface* relation, defining when an SPL implements an SPLS.



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

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

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

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

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

```
class FinAccount extends Object { // Base program
  String identity; double liquidity;
}                                             // Deltas
delta dSigPortfolio{
  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); }}
```

Fig. 5. Left: CapAccInt SPLS: $\mathcal{M}_{\mathsf{CapAccInt}}$ (top), $\mathcal{K}_{\mathsf{CapAccInt}}$ (middle), and $ABS_{\mathsf{CapAccInt}}$ (bottom). This SPLS is an interface of the CapitalAccount SPL if Figure 3 (left). It hides features BalanceInfo and YearlyFees. **Right:** FinAccInt SPLS: $\mathcal{M}_{\mathsf{FinAccInt}}$ (top), $\mathcal{K}_{\mathsf{FinAccInt}}$ (middle), and $AB_{\mathsf{FinAccInt}}$ (bottom). This SPLS is an interface of the FinancialAccount SPL of Figure 3 (right). It hides feature AmountInfo.

Definition 6 (Generator of an SPLS). *The* generator *of an SPLS* $\mathtt{Z}$*, denoted by* $\mathcal{G}_{\mathtt{Z}}$*, is a mapping from products to program signatures defined similarly to the generator of an SPL (see Definition 3).*

Definition 7 (Program interface). *A program signature* PS_{Int} *is an* interface *of program* P*, denoted as* $PS_{Int} \preceq P$*, iff* PS_{Int} *is obtained from* P *by dropping some class or attributes, the body of the remaining methods and by replacing some* **extends** $\mathtt{C}$ *clause by* **extends** $\mathtt{C}'$ *where* $\mathtt{C}'$ *is a superclass of* $\mathtt{C}$*.*

Definition 8 (SPL interface). *An SPLS* $\mathtt{Z}_{Int}$ *is an* interface *of an SPL* $\mathtt{L}$*, denoted as* $\mathtt{Z}_{Int} \preceq \mathtt{L}$*, iff: (i)* $\mathcal{M}_{\mathtt{Z}_{Int}} \preceq \mathcal{M}_{\mathtt{L}}$*; and (ii) the generators* $\mathcal{G}_{\mathtt{Z}_{Int}}$ *and* $\mathcal{G}_{\mathtt{L}}$ *are total and for each* $p \in \mathcal{P}_{\mathtt{L}}$*,* $\mathcal{G}_{\mathtt{Z}_{Int}}(p \cap \mathcal{F}_{\mathtt{Z}_{Int}}) \preceq \mathcal{G}_{\mathtt{L}}(p)$*.*

We say that an SPL $\mathtt{L}$ *implements* an SPLS $\mathtt{Z}$ when $\mathtt{Z}$ is an interface of $\mathtt{L}$.

Figure 5 represents an interface of SPL CapitalAccount (CapAccInt, on the left) and an interface of SPL FinancialAccount (FinAccInt, on the right), explanations are given in the caption.

3.2 Dependent SPLs

A *Dependent SPL* (DPL) is an SPL extended with dependencies modeled by SPLSs. The abstract syntax of IFMΔJ DPLs is given in Figure 4 (bottom). 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 $\mathcal{M}_{Glue}$, configuration knowledge $\mathcal{K}$ and an artifact base AB. The two feature models $\mathcal{M}_{Main}$ and $\mathcal{M}_{Glue}$ structure the actual feature model $\mathcal{M}_{\mathtt{L}}$ of L in two parts: $\mathcal{M}_{Main}$ describes the part of $\mathcal{M}_{\mathtt{L}}$ that is local to L, while $\mathcal{M}_{Glue}$ states how the features of $\mathcal{M}_{\mathtt{L}}$ are related with the features of L's dependencies. Formally, the feature model of L is defined as a composition of $\mathcal{M}_{Main}$ and the feature models $\mathcal{M}_{\mathtt{Z}_1}, \dots, \mathcal{M}_{\mathtt{Z}_n}$, glued together with $\mathcal{M}_{Glue}$: $\mathcal{M}_{\mathtt{L}} = \mathcal{M}_{Main/\overline{\mathtt{Z}}} = \mathcal{M}_{Main} \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{\overline{\mathtt{Z}}}$ where $\mathcal{M}_{\overline{\mathtt{Z}}} = \mathcal{R}(\mathcal{M}_{\mathtt{Z}_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{\mathtt{Z}_n})$. Lemma 1 below guarantees that the order of $\mathtt{Z}_1$,. . . ,$\mathtt{Z}_n$ is immaterial.

Lemma 1 (Join operation). *The join operation $\bullet$ is associative and commutative, with $\mathcal{M}_{Id} = \mathcal{R}(\mathcal{M}_{\emptyset}) = \mathcal{R}((\emptyset, \emptyset)) = (\emptyset, \{\emptyset\})$ as identity.*

Figure 6 presents the DPL DualAccount with dependencies CapAccInt and FinAccInt—explanations are given in the caption.

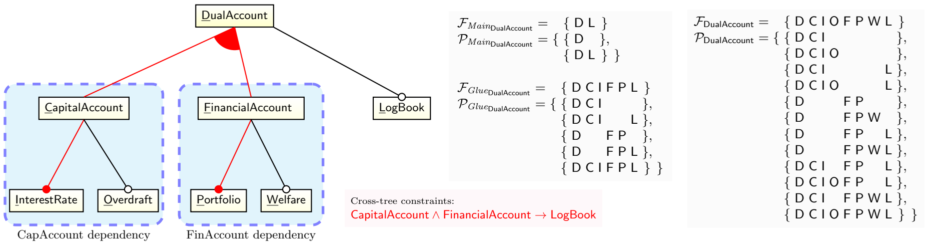
Remark 3 (DPL conservatively extends SPL). In order to ensure that the concept of DPL is a conservative extension of the concept of SPL (cf. Section 2.1), we assume that if a DPL L has no dependencies (i.e., $\overline{\mathtt{Z}} = \emptyset$) then $\mathcal{M}_{Glue} = \mathcal{M}_{Id}$ (cf. Lemma 1). Therefore: (i) any DPL L without dependencies can be seen as an SPL with feature model $\mathcal{M}_{\mathtt{L}} = \mathcal{M}_{Main}$; and (ii) any SPL L can be seen as a DPL with $\mathcal{M}_{Main} = \mathcal{M}_{\mathtt{L}}$ and $\mathcal{M}_{Glue} = \mathcal{M}_{Id}$.

Definition 9 (Multi Software Product Lines). *A* Multi Software Product Line *(MPL) is a set of SPL Signatures and Dependent SPLs.*

Sanity Conditions. To simplify the manipulation of our model in the rest of the document, we give here a set of standard sanity conditions that are supposed to be satisfied by the MPLs that we consider in this paper. First, we suppose that all the DPL and SPLS names used in an MPL are declared exactly once in the MPL. Second, we suppose that a DPL depends *only once* on an SPLS, i.e., the list of dependencies ($\overline{\mathtt{Z}}$) in the DPL syntax does not contain duplicates. Finally, we suppose that a class can only be declared and modified by at most one DPL in an MPL. Note that class disjointness enforces a boundary between different DPLs and rules out class name clashes between variants of different DPLs. Moreover, 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., each delta name may belong to a unique DPL or SPLS).

3.3 DPLs Composition

The concept of *DPL-SPLs composition* formalizes composition of software product lines through aggregation by means of the concepts of DPL and SPL interface (i.e., by inclusion of some SPLs into a DPL to fulfill its dependencies)—thus extending the concept of feature model composition to encompass the configuration knowledge and the artifact base.



Cross-tree constraints:
CapitalAccount ∧ FinancialAccount → LogBook

$$\begin{array}{ll}
\mathcal{F}_{Main_{\mathsf{DualAccount}}} = & \{\,\mathsf{D\ L}\,\} \\
\mathcal{P}_{Main_{\mathsf{DualAccount}}} = & \{\,\{\,\mathsf{D}\,\},\ \{\,\mathsf{D\ L}\,\}\,\}
\end{array}$$

$$\begin{array}{ll}
\mathcal{F}_{Glue_{\mathsf{DualAccount}}} = & \{\,\mathsf{D\ C\ I\ F\ P\ L}\,\} \\
\mathcal{P}_{Glue_{\mathsf{DualAccount}}} = & \{\,\{\,\mathsf{D\ C\ I}\,\},\ \{\,\mathsf{D\ C\ I\ L}\,\},\ \{\,\mathsf{D\ F\ P}\,\},\ \{\,\mathsf{D\ F\ P\ L}\,\},\ \{\,\mathsf{D\ C\ I\ F\ P\ L}\,\}\,\}
\end{array}$$

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

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

```
class DualAccount extends Object {String identity; void setId(String id){identity=id};} // Base program
delta dDualC { modifies class DualAccount extends Object {                              // Deltas
                  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, Date e){fin.portfolio.addToPortfolio(i,e);}}}
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, Date e){ journalLog+= "::add2P("+i+","+e+")"; original(i, e);}}}
delta dLogW { modifies class DualAccount extends Object {
     modifies void add2W(String i, String b){ journalLog+= "::add2W("+i+","+b+")"; original(i, b); }}}
```

Fig. 6. DualAccount DPL is declared as: **line** DualAccount(CapAccInt,FinAccInt) $\{\mathcal{M}_{Main_{\mathsf{DualAccount}}} \mathcal{M}_{Glue_{\mathsf{DualAccount}}} \mathcal{K}_{\mathsf{DualAccount}} AB_{\mathsf{DualAccount}}\}$. It has feature model $\mathcal{M}_{\mathsf{DualAccount}} = \mathcal{M}_{Main_{\mathsf{DualAccount}}/\mathsf{CapAccInt,FinAccInt}} = \mathcal{M}_{Main_{\mathsf{DualAccount}}} \circ \mathcal{M}_{Glue_{\mathsf{DualAccount}}} \mathcal{M}_{\mathsf{CapAccInt,FinAccInt}}$ (depicted as a feature diagram at the top of the figure); configuration knowledge $\mathcal{K}_{\mathsf{DualAccount}}$ (middle); and artifact base $AB_{\mathsf{DualAccount}}$ (bottom). It provides a class `DualAccount` that combines two bank accounts that satisfy the dependencies CapAccInt and FinAccInt (given in Figure 5), respectively. The feature model $\mathcal{M}_{\mathsf{DualAccount}}$ is the composition of four feature models. (i) The feature model $\mathcal{M}_{Main_{\mathsf{DualAccount}}}$, which comprises the mandatory feature DualAccount and the optional feature LogBook (that ensures that transactions are traced). (ii)-(iii) The feature models of the dependencies CapAccInt and FinAccInt (given in Figure 5). (iv) The feature model $\mathcal{M}_{Glue_{\mathsf{DualAccount}}}$, which has features DualAccount, LogBook, CapitalAccount,FinancialAccount, InterestRate, Portfolio and expresses the constraints FinancialAccount ∨ CapitalAccount, CapitalAccount → InterestRate FinancialAccount → Portfolio (represented by the parts colored in red of the feature diagram) and CapitalAccount ∧ FinancialAccount → LogBook (represented by the cross-tree constraint, also colored in red). The dashed rectangles depict the feature diagrams representing the feature model obtained from $\mathcal{M}_{\mathsf{CapAccInt}}$ and $\mathcal{M}_{\mathsf{FinAccInt}}$ by adding the constraints provided by the feature model $\mathcal{M}_{Glue_{\mathsf{DualAccount}}}$, respectively.

Definition 10 (DPL-SPLs composition). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ $(n \geq 0)$ *and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ *be SPLs such that* $\mathtt{Z}_i \preceq \mathtt{L}_i$ $(1 \leq i \leq n)$. *The composition of* $\mathtt{L}$ *with* $\overline{\mathtt{L}}$ *is the SPL (cf. Remark 3)* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ *such that:*[1]

[1] Because of the delta scope assumption, in the definition of $\mathcal{K}_{\mathtt{L}_0}$ the union of the application ordering relations (which denotes the relation obtained by union of their graphs) is well defined.

- $\mathcal{M}_{Main_{\mathtt{L}_0}} = \mathcal{M}_{Main_{\mathtt{L}}/\overline{\mathtt{L}}} = \mathcal{M}_{Main_{\mathtt{L}}} \circ_{\mathcal{M}_{Glue_{\mathtt{L}}}} \mathcal{M}_{\overline{\mathtt{L}}}$;
- $\mathcal{K}_{\mathtt{L}_0} = (\alpha_{\mathtt{L}_0}, <_{\mathtt{L}_0}) = (\alpha'_{\mathtt{L}} \cup \left(\bigcup_{i \in \{1,\ldots,n\}} \alpha'_{\mathtt{L}_i}\right)), <_{\mathtt{L}} \cup \left(\bigcup_{i \in \{1,\ldots,n\}} <_{\mathtt{L}_i}\right))$ *where*
  - $\alpha'_{\mathtt{L}}(\mathtt{d}) = \{p \in \mathcal{P}_{\mathtt{L}_0} \mid p \cap \mathcal{F}_{\mathtt{L}} \in \alpha_{\mathtt{L}}(\mathtt{d})\}$ *for all deltas* $\mathtt{d}$ *of* $\mathtt{L}$*;*
  - $\alpha'_{\mathtt{L}_i}(\mathtt{d}) = \{p \in \mathcal{P}_{\mathtt{L}_0} \mid p \cap \mathcal{F}_{\mathtt{L}_i} \in \alpha_{\mathtt{L}_i}(\mathtt{d})\}$ *for all deltas* $\mathtt{d}$ *of* $\mathtt{L}_i$*;*
- $AB_{\mathtt{L}_0} = AB_{\mathtt{L}} \cup \left(\bigcup_{i \in \{1,\ldots,n\}} AB_{\mathtt{L}_i}\right)$*; and*
- $\mathcal{M}_{Glue_{\mathtt{L}_0}} = \mathcal{M}_{Id}$.

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). For example, the DPL DualAccount can be composed with the SPLs CapitalAccount and FinancialAccount to obtain the SPL DualAccount(CapitalAccount,FinancialAccount).

The following theorems shed light on DPL-SPLs composition. Theorem 1 states that the variants of the composed SPL $\mathtt{L}(\overline{\mathtt{L}})$ can be generated by building the composed feature model $\mathcal{M}_{\mathtt{L}(\overline{\mathtt{L}})}$ and then using the generators of the DPL $\mathtt{L}$ and of the SPLs $\overline{\mathtt{L}}$—thus, there is no need to actually build the whole $\mathtt{L}(\overline{\mathtt{L}})$. Theorem 2 states that fulfilling the dependencies of a DPL preserves the set of implemented interfaces.

Theorem 1 (Generator of the composed product line). *Let* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$*. For each product* $p \in \mathcal{P}_{\mathtt{L}_0}$, $\mathcal{G}_{\mathtt{L}_0}(p) = \mathcal{G}_{\mathtt{L}}(p \cap \mathcal{F}_{\mathtt{L}}) \cup \left(\bigcup_{\mathtt{L}_i \in \overline{\mathtt{L}}} \mathcal{G}_{\mathtt{L}_i}(p \cap \mathcal{F}_{\mathtt{L}_i})\right)$.

Theorem 2 (DPL-SPLs composition preserves interfacing). *Let* $\mathtt{Z}$ *be an SPLS,* $\mathtt{L}$ *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 SPLs. If* $\mathtt{Z} \preceq \mathtt{L}$ *and* $\mathtt{Z}_i \preceq \mathtt{L}_i$ $(1 \leq i \leq n)$*, then* $\mathtt{Z} \preceq \mathtt{L}(\overline{\mathtt{L}})$.

In the following, we show that composition can also be done between DPLs: we just need to define the *interface* relation on DPLs and then extend the DPL-SPLs composition to DPL-DPL as well.

Definition 11 (DPL interface). *An SPLS* $\mathtt{Z}_{Int}$ *is an* interface *of an DPL* $\mathtt{L}$ *with dependencies* $\overline{\mathtt{Z}}$*, denoted as* $\mathtt{Z}_{Int} \preceq \mathtt{L}$*, iff (i)* $\mathcal{M}_{\mathtt{Z}_{Int}} \preceq \mathcal{M}_{\mathtt{L}}$*; and (ii) the generators* $\mathcal{G}_{\mathtt{Z}_{Int}}$*,* $\mathcal{G}_{\mathtt{L}}$ *and* $\mathcal{G}_{\overline{\mathtt{Z}}}$ *are total and for each* $p \in \mathcal{P}_{\mathtt{L}}$*,* $\mathcal{G}_{\mathtt{Z}_{Int}}(p \cap \mathcal{F}_{\mathtt{Z}_{Int}}) \preceq \bigcup \mathcal{G}^{\star}_{\mathtt{Z}}(p \cap \mathcal{F}_{\mathtt{Z}}) \cup \mathcal{G}_{\mathtt{L}}(p)$*, where* $\mathcal{G}^{\star}_{\mathtt{Z}}(p \cap \mathcal{F}_{\mathtt{Z}})$ *is equal to* $\mathcal{G}_{\mathtt{Z}}(p \cap \mathcal{F}_{\mathtt{Z}})$ *with all method declarations extended with the body* {**return null**;}.

The following definition extends the concept of DPL-SPLs composition (Definition 10) by accepting DPLs as arguments and yielding a DPL as result.

Definition 12 (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* $\overline{\mathtt{L}} = \mathtt{L}_1, \ldots, \mathtt{L}_n$ *be DPLs such that* $\mathtt{Z}_i \preceq \mathtt{L}_i$ $(1 \leq i \leq n)$*. Let* $\overline{\mathtt{Z}}^{(i)} = \mathtt{Z}_{i,1}, \ldots, \mathtt{Z}_{i,n_i}$ $(n_i \geq 0)$ *be the dependencies of* $\mathtt{L}_i$ $(1 \leq i \leq n)$*. The composition of* $\mathtt{L}$ *with* $\overline{\mathtt{L}}$ *is the DPL* $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$*, with dependencies* $\overline{\mathtt{Z}}^{(1)}, \ldots, \overline{\mathtt{Z}}^{(n)}$*, such that* $\mathcal{M}_{Main_{\mathtt{L}_0}}$*,* $\mathcal{K}_{\mathtt{L}_0}$ *and* $AB_{\mathtt{L}_0}$ *are defined as in Definition 10, and* $\mathcal{M}_{Glue_{\mathtt{L}_0}}$ *is defined by* $\mathcal{M}_{Glue_{\mathtt{L}_0}} = \mathcal{M}_{Glue_{\mathtt{L}_1}} \bullet \cdots \bullet \mathcal{M}_{Glue_{\mathtt{L}_n}}$.

Note that, if the DPLs $\mathtt{L}_i$ $(1 \leq i \leq n)$ have no dependencies (i.e., $\overline{\mathtt{Z}}^{(i)} = \emptyset$ and $\mathcal{M}_{Glue_{\mathtt{L}_i}} = \mathcal{M}_{Id}$), then $\mathcal{M}_{Glue_{\mathtt{L}_0}} = \mathcal{M}_{Id}$ (like in Definition 10). Thus Definition 12 conservatively extends Definition 10. Moreover, Theorem 1 also holds when $\overline{\mathtt{L}}$ and $\mathtt{L}_0 = \mathtt{L}(\overline{\mathtt{L}})$ are DPLs, and Theorem 2 can be extended as follows:

Theorem 3 (DPL-DPLs composition preserves interfacing). *Let* $\mathtt{Z}$ *be an SPLS,* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ *(*$n \geq 0$*), and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ *be DPLs. If* $\mathtt{Z} \preceq \mathtt{L}$ *and* $\mathtt{Z}_i \preceq \mathtt{L}_i$ *(*$1 \leq i \leq n$*), then* $\mathtt{Z} \preceq \mathtt{L}(\overline{\mathtt{L}})$*.*

4 Compositionality of Existing SPL Analyses

In this section, we give two initial results illustrating the fact that our MPL model is well-suited for compositional analysis. First, we show that the results about the compositionality of existing analyses of feature models (*void feature model*, *core features*, *dead features*, *void partial configuration*, and *atomic sets*) given in [24, Sect. 5] can be used as-is in our model. Second, we show how to extend existing type systems for SPLs to ensure well-typedness in our model.

Compositional Analysis of Feature Models. The following theorem shows that the construction of the feature model of a DPL can be expressed as a sequence of $\circ$ operations. This, plus the fact that an SPLS $\mathtt{Z}$ is an interface of a DPL $\mathtt{L}$ only when $\mathcal{M}_{\mathtt{Z}} \preceq \mathcal{M}_{\mathtt{L}}$ ensures that the results presented in [24, Sect. 5] can be used as-is to analyse the feature models constructed in DPL-DPL compositions by analysing each feature model independently.

Theorem 4. *Let* $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$*,* $\mathcal{M}_{y_1} = (\mathcal{F}_{y_1}, \mathcal{P}_{y_1})$*,...,*$\mathcal{M}_{y_n} = (\mathcal{F}_{y_n}, \mathcal{P}_{y_n})$*, with* $n \geq 1$*, be feature models with pairwise feature disjointness (cf. Remark 2) and* $\mathcal{M}_{\overline{y}} = \mathcal{R}(\mathcal{M}_{y_1}) \bullet \cdots \bullet \mathcal{R}(\mathcal{M}_{y_n})$*. Then (for every permutation* $w_1, ..., w_n$ *of* $y_1, ..., y_n$*):* $\mathcal{M}_{x/\overline{y}} = \mathcal{M}_x \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{\overline{y}} = ((\mathcal{M}_x \circ_{\mathcal{M}_{Id}} \mathcal{M}_{w_1}) \cdots \circ_{\mathcal{M}_{Id}} \mathcal{M}_{w_{n-1}}) \circ_{\mathcal{M}_{Glue}} \mathcal{M}_{w_n}$*.*

Compositional Type System for MPLs. Type checking an SPL means to check that all its variants can be generated and are well-typed programs. Performing this check by generating each variant and type checking it does not scale (a product line with n features can have up to 2^n products). Therefore, several SPL type checking approaches have been proposed in the literature [27]. Three type checking approaches for delta-oriented SPLs have been proposed and formalized [9,5,8] by means of the IFΔJ calculus.

In our MPL model, we add two structures that can be type-checked: DPLs and DPL-DPL compositions. However, due to the fact that the artifact base of a DPL depends on code defined in other DPLs, it is too restrictive to require that its variants are well-typed programs: they can indeed contain missing dependencies. The following definition extends the notion of well-typedness to DPL to deal with the missing dependency problem:

Definition 13 (Well-typed DPL). *The* stub-completion *of an SPLS* $\mathtt{Z}$*, written* $\mathtt{Z}^\star$*, is the SPL obtained by adding the body* {**return null**;} *to all the method declarations in* $\mathtt{Z}$*. The* stub-completion *of a DPL* $\mathtt{L}$ *with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ *(*$n \geq 0$*) is the SPL* $\mathtt{L}^\star = \mathtt{L}(\mathtt{Z}_1^\star, ..., \mathtt{Z}_n^\star)$ *obtained by composing* $\mathtt{L}$ *with the stub-completion of its dependencies. We say that a DPL is* well-typed *iff its stub-completion is well-typed.*

Note that this definition generalizes the notion of well-typedness for SPLs: when the set of dependencies of the DPL $\mathtt{L}$ is empty ($n = 0$), $\mathtt{L}$ is well-typed iff it is well-typed in the SPL-sense of the term. Moreover, with this definition, extending the exisiting type-checking algorithms for SPL to manage DPL simply requires a pre-processing of the DPL to transform it in an SPL as described in the definition. An additional important property of this definition is that it is enough to type-check in isolation the DPLs in a DPL-DPL composition to ensure that the resulting DPL is well-typed:

Theorem 5 (Compositionality of DPL-DPLs composition type checking). *Let* $\mathtt{L}$ *be a DPL with dependencies* $\overline{\mathtt{Z}} = \mathtt{Z}_1, ..., \mathtt{Z}_n$ *(*$n \geq 0$*) and* $\overline{\mathtt{L}} = \mathtt{L}_1, ..., \mathtt{L}_n$ *be DPLs such that* $\mathtt{Z}_i \preceq \mathtt{L}_i$ *(*$1 \leq i \leq n$*). If each of the DPLs* $\mathtt{L}, \mathtt{L}_1, ..., \mathtt{L}_n$ *type checks, then* $\mathtt{L}(\mathtt{L}_1, ..., \mathtt{L}_n)$ *type checks.*

Note that the SPLs CapitalAccount and FinancialAccount (in Figure 3), and the DPL DualAccount (in Figure 6) type check: we can then conclude that the SPL DualAccount(CapAccount,FinAccount) type checks as well.

Checking the Interface Relation. The compositional analysis of feature models and the well-typedness of a DPL-DPL composition $\mathtt{L}(\overline{\mathtt{L}})$ presented previously heavily rely on the interface relation being satisfied between the dependencies of $\mathtt{L}$ and the DPLs $\overline{\mathtt{L}}$. It is possible to automatically check this relation between any SPLS $\mathtt{Z}$ and any DPL $\mathtt{L}$ using a predicate formula written $\mathtt{match}(\mathtt{Z}, \mathtt{L})$. Due to lack of space, we cannot give the definition of this formula, we simply state the following theorem:

Theorem 6 (DPL interface checking). *If the SPLS* $\mathtt{Z}$ *and DPL*$\mathtt{L}$ *type check and* $\mathcal{M}_{\mathtt{Z}} \preceq \mathcal{M}_{\mathtt{L}}$ *holds, then* $\mathtt{match}(\mathtt{Z}, \mathtt{L})$ *is valid if and only if* $\mathtt{Z} \preceq \mathtt{L}$ *holds.*

5 Related Work and Conclusions

An extension of DOP to implement MPLs has been outlined in [10] 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 is very flexible, but it does not enforce any boundary between different SPLs—thus providing no support for compositional analyses.

Schröter et al. [25] advocated investigating suitable interfaces in order to support compositional analyses of MPLs for different stages of the development process. In particular, *syntactical interfaces*, which build on feature model interfaces to provide a view of reusable programming artifacts, and *behavioral interfaces*, which in turn build on syntactical interfaces to support formal verification. More recently, Schröter et al. [24] 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 build on [24] and propose the concepts of SPLS, DPL, and DPL-DPLs composition and show how to use them to support compositional type checking of delta-oriented MPL. An SPL signature is 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 [26] are aimed at supporting type checking SPLs developed according to the FOP approach which, as pointed out in Section 1, is encompassed by DOP (see [22] for a detailed comparison between FOP and DOP). A feature-context interface supports type checking a feature module 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. [16] 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): [16] considers an *annotative approach* (`#ifdef` preprocessor directives), while we consider a *transformational approach* (DOP)—we refer to [23,27] for classification and survey of different approaches for implementing variability.

Schröter et al. [24] defined a slice function for feature models (similar to the operator proposed by Acher et al. [1]) that generates a feature-model interface by removing a given set of features. In future work we would like to generalize the slice function for feature models to DPLs, thus providing an automatic means for generating an interface for a given DPL.

Recently, Thüm et al. [28] proposed a notion of behavioral interface for supporting compositional verification of FOP SPLs via variability encoding [29]. 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 [11,12,6].

We plan to implement our approach for both DeltaJ 1.5 [17] (a prototypical implementation of DOP that supports full Java 1.5) and the Abstract Behavioral Specification modeling language [15].

Acknowledgments. We thank the anonymous reviewers for comments and suggestions for improving the presentation. We also thank Lorenzo Testa for comments and suggestions during the preparation of the post-proceedings version.

References

1. M. Acher, P. Collet, P. Lahire, and R. B. France. Slicing feature models. In *26th IEEE/ACM International Conference on Automated Software Engineering, (ASE), 2011*, pages 424–427, 2011. DOI: 10.1109/ASE.2011.6100089.
2. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.
3. D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of International Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005. DOI: 10.1007/11554844_3.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
5. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
6. R. Bubel, F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, and 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*, 1:130–156, 2016. DOI: 10.1007/978-3-319-46508-1_8.
7. P. Clements and L. Northrop. *Software Product Lines: Practices & Patterns.* Addison Wesley Longman, 2001.
8. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In *Integrated Formal Methods: 12th International Conference, iFM 2016*, volume 9681 of *LNCS*, pages 47–62. Springer, 2016. DOI: 10.1007/978-3-319-33693-0_4.
9. F. Damiani and 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*, volume 7609 of *LNCS*, pages 193–207. Springer, 2012. DOI: 10.1007/978-3-642-34026-0_15.
10. F. Damiani, I. Schaefer, and T. Winkelmann. Delta-oriented multi software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 232–236. ACM, 2014. DOI: 10.1145/2648511.2648536.
11. R. Hähnle and 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*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012. DOI: 10.1007/978-3-642-34026-0_4.
12. R. Hähnle, I. Schaefer, and R. Bubel. Reuse in software verification by abstract method calls. In *Proceedings of International Conference on Automated Deduction*, CADE'13, pages 300–314. Springer, 2013. 10.1007/978-3-642-38574-2_21.
13. G. Holl, P. Grünbacher, and R. Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Information & Software Technology*, 54(8):828–852, 2012.
14. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012. DOI: 10.1007/978-3-642-25271-6_8.
16. C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming*

Systems Languages and Applications, OOPSLA '12, pages 773–792. ACM, 2012. DOI: 10.1145/2384616.2384673.

17. J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and 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*, pages 63–74, 2014. DOI: 10.1145/2647508.2647512.
18. M. Lienhardt and 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*, pages 178–192, 2012. DOI: 10.1007/978-3-642-34026-0_14.
19. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.
20. M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and 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, pages 13–18. Department of Informatics and Mathematics, University of Passau, 2008.
21. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (SPLC 2010)*, volume 6287 of *LNCS*, pages 77–91, 2010. DOI: 10.1007/978-3-642-15579-6_6.
22. I. Schaefer and F. Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 49–56. ACM, 2010. 10.1145/1868688.1868696.
23. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
24. R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and 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, pages 667–678. ACM, 2016. DOI: 10.1145/2884781.2884823.
25. R. Schröter, N. Siegmund, and T. Thüm. Towards modular analysis of multi product lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC'13, pages 96–99. ACM, 2013. DOI: 10.1145/2499777.2500719.
26. R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-context interfaces: Tailored programming interfaces for spls. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC'14, pages 102–111. ACM, 2014. DOI: 10.1145/2648511.2648522.
27. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.
28. T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, and 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, pages 97–104. ACM, 2016. DOI: 10.1145/2866614.2866628.
29. A. von Rhein, T. Thm, I. Schaefer, J. Liebig, and S. Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 2):125–145, 2016.