# Same Same But Different: Interoperability of Software Product Line Variants*

Ferruccio Damiani[1], Reiner Hähnle[2],
Eduard Kamburjan[2], and Michael Lienhardt[1]

**Abstract** Software Product Lines (SPLs) are an established area of research providing approaches to describe multiple variants of a software product by representing them as a highly variable system. Multi-SPLs (MPLs) are an emerging area of research addressing approaches to describe sets of interdependent, highly variable systems, that are typically managed and developed in a decentralized fashion. Current approaches do not offer a mechanism to manage and orchestrate multiple variants from one product line within the same application. We experienced the need for such a mechanism in an industry project with Deutsche Bahn, where we do not merely model a highly variable system, but a system with highly variable subsystems. Based on MPL concepts and delta-oriented oriented programming, we present a novel solution to the design challenges arising from having to manage and interoperate multiple subsystems with multiple variants: how to reference variants, how to avoid name or type clashes, and how to keep variants interoperable.

## 1 Introduction

Many existing software and non-software systems are built as complex assemblages of highly variable subsystems that coexist in multiple variants and that need to interoperate. Consider, for example, one track in a railway system. Such a track typically contains many different variants of sensors (to detect a train, its speed, etc.), and many variants of signals (of different forms or functions). The `FormbaR`[2] modeling project, conducted with Deutsche Bahn, aims to provide a uniform and formal

University of Torino, 10124 Torino, Italy e-mail: ferruccio.damiani@unito.it, michael.lienhardt@di.unito.it · Technische Universität Darmstadt, 64289 Darmstadt, Germany e-mail: {haehnle, kamburjan}@cs.tu-darmstadt.de

* This paper is dedicated to our friend and colleague Arnd Poetzsch-Heffter on the occasion of his sixtieth birthday.

[2] https://formbar.raillab.de

model [17] of operational and technical rulebooks for railroad operations: within this project, the necessity to describe highly variable subsystems that coexist in multiple variants and that need to interoperate thus arises naturally.

In software systems, there exist several approaches to model highly variable systems, labeled as *Software Product Lines* (SPLs) [5, 19, 2, 22]. These are generalized by *Multi-Software Product Lines* (MPLs) [14, 10] that model sets of interdependent highly variable systems, typically managed and developed in a decentralized fashion by multiple stakeholders. However, MPLs do not target modeling of interoperability between multiple variants of the same SPL.

To address this issue we introduce the notion of *variant-interoperable SPL* (VPL) and propose linguistic mechanisms that support interoperability among different variants of one VPL. Each VPL encapsulates and models the variability of one system. We define a formalism that is able to reference, to generate and to compose multiple variants of one VPL in the context of its supersystem. To do so, each variant is associated with one (possibly newly generated) module and statements are able to use *variant references* instead of modules to reference classes and interfaces. During variant generation, all such variant references are replaced by the module which contains the generated variant. The final variant of the whole system contains no SPL-specific constructs. We also give a generalization of VPLs to *dependent VPLs* (DVPL). A DVPL takes variants of other product lines as parameters and is thus able to model the *composition* of variable subsystems. A VPL is obtained as the special case of a DVPL without parameters. Thus, in our approach, an MPL can be described by: (i) a set of DVPLs; and (ii) a *glue program* that may contain references to different variants of the DVPLs.

*Delta-Oriented Programming* (DOP) [20] is a flexible and modular approach to implement SPLs. A delta-oriented 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* comprising a *base program* and of a set of *delta modules* (*deltas* for short), which are containers of program modifications (e.g., for Java-like programs, a delta can add, remove or modify classes and interfaces); and (iii) *configuration knowledge* which defines how to generate the SPL's variants by specifying an *activation mapping* that associates 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: given a product the corresponding variant is derived by applying the activated deltas to the base program according to the application ordering. DOP is a generalization of *Feature-Oriented Programming* (FOP) [3], a previously proposed approach to implement SPLs where deltas correspond one-to-one to features and do not contain remove operations.

In the context of the `FormbaR` project we model railway operations [17] using the Abstract Behavioural Specification (ABS) [13, 16] language, a delta-oriented modeling language. The challenge to model interoperable, multiple variants of the same subsystem that arose in this project is described in [7]. In ABS, variants are expressed in the executable language fragment Core ABS [16]. In this paper, we use railway *stations* and *signals* as a running example. We use a non-dependent VPL to model signals (which may be light or form signals and main or pre signals). A station is a DVPL that takes two signal SPLs variants as input. We illustrate the modeling capabilities by showing how one can ensure that all signals of a station are either light signals or all are form signals (generalizing the treatment of features from [10]). We also show how to model that every main signal is preceded by a pre signal.

Our contribution is the design of a delta-oriented DVPL language that can model interoperation of *multiple* variants from the *same* product line as well as from *different* product lines. We do not aim to fully explore the design space, but provide a concise system for basic functionality. However, we provide a discussion of our design decisions and how interoperability changes the role of product lines during development.

This work is structured as follows: Sect. 2 introduces FAM (Featherweight Core ABS with Modules) a foundational language for Core ABS. Sect. 3 introduces delta-oriented (non-dependent) VPLs on top of FAM. Sect. 4 generalizes to dependent interoperable product lines. Sect. 5 gives the reasoning behind our design decisions and how interoperability affects modeling. Sect. 6 gives related work and Sect. 7 concludes.

## 2 Featherweight Core ABS with Modules

In this section we introduce FAM (Featherweight Core ABS with Modules) a foundational language for Core ABS [16]. Following [15], we use the overline notation for (possibly empty) sequences of elements. For example, $\overline{\mathsf{CD}}$ stands for a sequence of class declarations $\mathsf{CD}_1 \cdots \mathsf{CD}_n$ $(n \geq 0)$—the empty sequence is denoted by $\emptyset$. Moreover, when no confusion may arise, we identify sequences of pairwise distinct elements with sets. We write $\overline{\mathsf{CD}}$ as short for $\{\mathsf{CD}_1 \ldots, \mathsf{CD}_n\}$, etc. FAM is an extension of Featherweight Core ABS [6], a previously proposed foundational language for Core ABS, that does not model modules. As seen in Sect. 3, modules play a key role in the definition of variants in VPL.

Fig. 1 shows the abstract syntax of FAM. A FAM program Prgm consists of a set of modules Mod. A module has a name M, import and export clauses, a set of class definitions CD and a set of interface declarations ID. To use a class defined in one module in a different module, the defining module must export it and the using module must import it. There are no such restrictions when using a class inside its defining module. We allow wildcards $\star$ in the import/export clauses.

A class definition CD consists of a name C, an optional **implements** clause and a set of method and field definitions. The references CR and IR are used respectively to reference classes and interfaces inside of modules. We assume some primitive types (including `Unit`, used as return type for methods without a return value) and let T range over interface names and primitive types.

Class definitions and interface definitions in ABS are similar to Java, but ABS does not support class inheritance. Our development is independent of the exact

| | |
|---|---|
| $\mathsf{Prgm} ::= \overline{\mathsf{Mod}}$ | Program |
| $\mathsf{Mod} ::= \textbf{module}\ \mathsf{M};\ \textbf{import}\ \mathsf{SC}\ \textbf{from}\ \mathsf{M};\ \textbf{export}\ \mathsf{SC};\ \overline{\mathsf{CD}}\ \overline{\mathsf{ID}}$ | Module |
| $\mathsf{SC} ::= \overline{\mathsf{C}},\overline{\mathsf{I}} \mid \star \qquad \mathsf{CD} ::= \textbf{class}\ \mathsf{C}\ [\textbf{implements}\ \mathsf{IR}\ \overline{\mathsf{IR}}]\ \{\overline{\mathsf{AD}}\}$ | Selection, Class |
| $\mathsf{CR} ::= \mathsf{M.C} \mid \mathsf{C} \qquad \mathsf{IR} ::= \mathsf{M.I} \mid \mathsf{I}$ | Class/Interface Reference |
| $\mathsf{AD} ::= \mathsf{FD} \mid \mathsf{MD} \qquad \mathsf{FD} ::= \mathsf{T}\ \mathsf{f}{=}\mathsf{e} \qquad \mathsf{MD} ::= \mathsf{MSD}\{\ldots\}$ | Attribute (Field, Method) |
| $\mathsf{MSD} ::= \mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}\ \mathsf{v}}) \qquad \mathsf{ID} ::= \textbf{interface}\ \mathsf{I}\ [\textbf{extends}\ \mathsf{IR}\ \overline{\mathsf{IR}}]\ \{\overline{\mathsf{MSD}}\}$ | Signature, Interface |
| $\mathsf{e} ::= \textbf{new}\ \mathsf{CR}(\overline{\mathsf{e}}) \mid \ldots \qquad \mathsf{T} ::= \mathsf{IR} \mid \mathtt{Unit} \mid \mathtt{Int} \mid \cdots$ | Expression, Type |

**Fig. 1** Syntax of Featherweight Core ABS with Modules (FAM)—expressions and statements (method bodies) are left unspecified.

syntax of expressions e and statements s, so we leave it unspecified. We show only the object creation expression **new** $\mathsf{CR}(\overline{\mathsf{e}})$, which creates a new instance of the class referenced by CR.

## 3 Delta-Oriented VPLs

We introduce Featherweight *Delta* ABS with Modules (FDAM), a foundational language for delta-oriented VPLs where variants are FAM programs. FDAM is an extension of Featherweight Delta ABS (FDABS) [6], a foundational language for standard ABS *without* variant interoperability.

### *3.1 Variant-Interoperable Product Lines*

To handle multiple variants of an SPL and ensure their interoperability, we need to introduce several mechanisms that extend FAM:

1. We must be able to reference different variants of the same SPL. To this end, a class reference may be prefixed with a *variant reference*, i.e. a syntactic construct that identifies a specific SPL variant.
2. The notion of artifact base of a delta-oriented SPL must support interoperability of different SPL variants: specifically, different variants must be able to share common interfaces. This is achieved with a **unique** block containing the code that is common to all variants.
3. The variant generation process must generate code that can coexist and interoperate, even though the variants will necessarily have overlapping signatures. To this end, the code of each referenced variant is encapsulated by placing it in a separate module, while variant references are replaced by module references. As

each variant refers to a unique module, multiple references to the same variant refer to the same module (that is, they are generated exactly once).

## 3.2 Syntax

*Featherweight Delta ABS with Modules* (FDAM) is a language for delta-oriented VPLs where variants are FAM programs. It allows to describe an MPL by a set of VPLs $\overline{\mathsf{Vpl}}$ and a *glue program* Gprgm (i.e. a program that may contain variant references). Fig. 2 gives the formal syntax of VPLs and *extended references* (i.e. the class/interface references allowed in the glue program, that may be prefixed variant references).

| | |
|---|---|
| Vpl ::= **productline** V; **features** $\overline{\mathsf{F}}$ **with** $\varphi$;<br>Prgm **unique** $\overline{\mathsf{Mod}}$ $\overline{\Delta}$ DConfig | VPL |
| $\Delta$ ::= **delta** D; $\overline{\mathsf{CO}}$ $\overline{\mathsf{IO}}$ | Delta |
| DConfig ::= $\overline{\mathsf{DAC}}$ DAC ::= **delta** D **when** $\varphi$; | Configuration Knowledge |
| CO ::= CAO \| CMO \| CRO \| **uses** M | Class Operation |
| CAO ::= **adds** CD CRO ::= **removes** CR | Class Add/Remove Operations |
| CMO ::= **modifies** CR{AO} | Class Modifies Operations |
| AO ::= **adds** AD \| **removes** MSD \| **removes** T f<br>**modifies** AD | Attribute Operation |
| IO ::= IAO \| IMO \| IRO \| **uses** M | Interface Operation |
| IAO ::= **adds** ID IRO ::= **removes** IR | Interface Add/Remove Operations |
| IMO ::= **modifies** IR{SO} | Interface Modify Operation |
| SO ::= **adds** MSD \| **removes** MSD | Signature Operation |

| | |
|---|---|
| CR ::= VR.M.C \| M.C \| C IR ::= VR.M.I \| M.I \| I | Extended Class/Interface Reference |
| VR ::= V \| V[$\overline{\mathsf{F}}$] | Variant Reference |

**Fig. 2** Syntax of Featherweight Delta ABS with Modules: VPLs (top) and extended references (bottom).

A Vpl has a unique name V, and a set of features $\overline{\mathsf{F}}$, which are constrained by some feature model $\varphi$, a propositional formula over $\overline{\mathsf{F}}$. Furthermore, a VPL has a set of deltas $\overline{\Delta}$ and configuration knowledge DConfig (comprising an ordered sequence of delta activation clauses DAC) that relates each delta to an activation condition and specifies a partial order of delta application. Finally, a VPL has a *base program* as well as a **unique** block, consisting of module definitions on which the deltas operate.

Each delta has a name D and a sequence of class/interface operations. A class/interface operation may add, modify or remove a class/interface. A **uses** clause sets a module name as default prefix for further selections. During variant generation the application of the delta throws an error if the element is already in the code (if it is supposed to be added) or absent (if it is supposed to be removed or modified).[3] Adding and removing a class/interface is straightforward. In case of class/interface modification, a delta may add or remove signatures in interfaces and attributes in classes. A class modification may also modify an attribute: either replace the initialization expression of a field or replace the body of a method (the new body can call the original implementation of the method with the keyword **original** [6]).

Within a glue program class/interface references have the possibility to reference a class/interface of a variant by extended references. An extended reference is a class/interface reference that may be prefixed by a variant reference. A variant reference VR consists either of the name of the target VPL V and the features $\overline{\mathsf{F}}$ used for variant selection; or simply the name of the target VPL V when selecting a **unique** class/interface. A variant is selected by providing a set of features $\overline{\mathsf{F}}$ to a VPL. If that set does not satisfy the feature model $\varphi$, then an error is thrown during variant generation. All other clauses are defined as in Sect. 2.

The only form of extended references allowed in a VPL V are of the form V.C or V.I to reference *its own* **unique** block. Variant selections and references to the **unique** part of other VPLs are not allowed.

Intuitively, the generation of the variants referenced from an FDAM glue program works as follows: A new module name `M` is created, and modules `mod` from the **unique** block are added under `M_mod`. Next, for each referenced variant, each configured delta is applied to a copy of the base program, provided its activation condition is satisfied. All modified classes/interfaces are copied into `M_mod` and modified there. All added classes/interfaces are added into `M_mod`. Finally, all references are updated and all variant references occurring in the glue program are replaced by references to the generated modules. Fig. 3 illustrates this workflow and we give a more detailed description in Sect. 3.4.

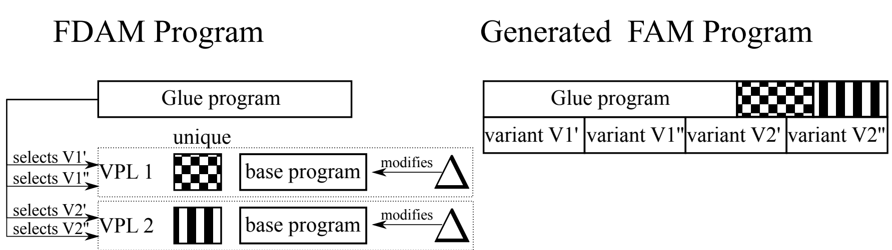


**Fig. 3** Schematic Overview over a FDAM program (representing an MPL) and the generated FAM Program.

[3] The ABS tool chain is equipped with a mechanism for statically detecting these errors [9].

### 3.3 A VPL for Railway Signals

We illustrate the VPL concept with a model of railway signals, see Fig. 4. A signal is either a main or a pre signal and either a form signal (showing its signal aspects with geometric shapes) or a light signal (using colors and light patterns). This is modeled by the features `Pre`, `Main`, `Light`, `Form`, respectively. We impose the constraint that exactly one of `Main` and `Pre` and one of `Form` and `Light` must be selected.

```
productline SLine;
features Main, Pre, Light, Form with Main↔¬Pre ∧ Light↔¬Form
 ;
module BMd;
class Signal implements SLine.SMd.Sig {}
unique{
  module SMd;
  interface Sig { ... }
}
delta SigForm;  modifies class BMd.Signal { ... } ...
delta SigPre;   modifies class BMd.Signal { ... } ...
delta SigMain;  modifies class BMd.Signal { ... } ...
delta SigLight; modifies class BMd.Signal { ... } ...

// Glue program
module main;
class Main{
  Unit main() {
    SLine.SMd.Sig s1 = new SLine[Pre,Form].BMd.Signal();
    SLine.SMd.Sig s2 = new SLine[Main,Form].BMd.Signal();
    s1.connect(s2);
    SLine.SMd.Sig s3 = new SLine[Pre,Form].BMd.Signal();
    SLine.SMd.Sig s4 = new SLine[Main,Form].BMd.Signal();
    s3.connect(s4);
  }
}
```

**Fig. 4** A VPL for railway signals (configuration knowledge, which associates an activation condition to each delta and specifies the application order of the the delta, is omitted) and a glue program that uses it.

The **unique** block provides an interface `SMd.Sig` which serves as the interface of the signal model to the outside. The base program provides an empty class `BMd.Signal` that implements this interface. Every variant of the VPL `SLine` generates a different variant of the class of `Signal` by adding the required functionality. We do not provide complete delta declarations. While we focus on the `Signal` class, each delta can add auxiliary classes (for example, a `Bulb` class for light signals).

The glue program contains the `main` module, providing the `Main` class with the `main()` method that creates a station with two main and two pre signals. After the declarations in the `main()` method, an expression like `s1 == s2` would type check. Observe that the `Signal` classes must be referenced with a variant selection, but this is not necessary for `Sig`, because it is **unique**. This is appropriate, because all `Signal` classes in all variants implement it. We provide a few examples to further illustrate the role of the **unique** block.

*Example 1 (Empty* **unique** *block).* Consider Fig. 4, but with an empty **unique** block. The interface is added to module `BMd` in the variants instead, and the first two lines of the main method are replaced with

```
18 SLine[Pre,Form].BMd.Sig  s1 = new SLine[Pre,Form].BMd.Signal
   ();
19 SLine[Main,Form].BMd.Sig s2 = new SLine[Main,Form].BMd.
   Signal();
```

Then `s1 == s2` would *not* type check, because each referenced type is added as a separate interface. However, if lines 21, 22 of Fig. 4 are changed accordingly, then `s1 == s3` *would* still type check, because the selected features identify a variant uniquely.

*Example 2 (Empty base program).* Consider again Fig. 4 but with an empty base program, where all deltas *modify* the interface and add classes that implement the modified interface. For example, replace **delta** `SigForm` with

```
delta SigForm; modifies interface SLine.SMd.Sig { ... }
adds class SMd.Signal implements SMd.Sig { ... }
```

In this case, line 18 of Fig 4 won't type check, because the `Sig` class of the variant is based on a copy of the non-variant class and is not its subtype.

### 3.4 Glue Program Flattening for FDAM

*Glue program flattening* refers to the transformation of an FDAM program that models an MPL, i.e. a glue program plus a set of VPLs, into an FAM program, see Fig. 3. This transformation involves code generation for all the variants referenced in the glue program (as outlined at the end of Sect. 3.2). Consider the MPL consisting of the glue program and the VPL in Fig. 4.

We assume an injective function *mod* mapping variant references and module names to fresh (relative to the glue program) names. We assume *mod* ignores the order of features. For each variant selection $\mathsf{V}(\overline{\mathsf{F}})$ and each module `mod` this function is used to create a new module with name $mod(\mathsf{V}(\overline{\mathsf{F}}),\mathtt{mod})$ and for each VPL $\mathsf{V}$ and each module `mod` it is used to create a new module with name $mod(\mathsf{V},\mathtt{mod})$.

*Example 3.* In Fig. 4, for each module `mod` in the **unique** block a module named $mod(\mathtt{SLine},\mathtt{mod})$ is created, to which the unique modules and classes are added. Next, each variant reference in the glue code is processed. Let us consider `SLine[Pre,Form].BMd`. The selected feature set is checked against the constraint of the

VPL. In this case, $\{$`Pre`,`Form`$\}$ satisfies `Main` $\leftrightarrow \neg$`Pre` $\wedge$ `Light` $\leftrightarrow \neg$`Form`. The configuration knowledge is used to determine which deltas are applied in which order to the base program. Here, only `SigPre` and `SigForm` are applied.

For each class/interface `M.C`/`M.I` added in any delta activated to generate the selected variant, a module *mod*(`SLine[Pre,Form]`),`M`]) is created (if it does not yet exist) and the class is added there. For each class/interface reference `mod.C′` in `M.C`, a clause **import** `C′` **from** `mod;` is added to *mod*(`SLine[Pre,Form]`,`M`). Finally, an **export** `*;` clause is added.

For each class/interface `M.C`/`M.I` modified in any delta activated to generate the selected variant, a module *mod*(`SLine[Pre,Form]`,`M`) is created (if it does not yet exist) and the class/interface is copied there *before* any modifications are applied. In this case, all **import** and **export** clauses are also copied from their original module.

During post-processing, all variant references `SLine[Pre,Form].M.C` are replaced by *mod*(`SLine[Pre,Form]`,`M`)`.C`. This reference is made visible by the clause **import** `C` **from** *mod*(`SLine[Pre,Form]`,`M`) added to the containing module.

This algorithm is applied recursively on the resulting program. If we apply the described algorithm once to the FDAM MPL in Fig. 4, then the FAM program in Fig. 5 is generated, where an obvious choice for *mod* has been adopted.

## 4 Delta-Oriented DVPLs

The VPL concept makes it possible to reference multiple variants of a product line from a glue program that is external to the product line. However, one has to know the exact product at each variant reference. If, for example, we attempt to model a station that has light signals as well as form signals, this leads to code duplication. This can be avoided by making VPLs parametric in the referenced variants: We extend VPLs to *dependent VPLs* (DVPL). A DVPL takes variants of other product lines as parameters: a product of a DVPL is identified by a set of features and a set of product lines (matching the parameters), each of them with an associated product.

### *4.1 Syntax*

We extend the FDAM language from Sect. 3 to *Featherweight Dependent Delta ABS with Modules* (FDDAM). Fig. 6 gives the formal syntax. Product lines are extended with optional product line parameters $\mathsf{P}$. These parameters can be used used in the feature model, which may reference features of the passed parameters with $\mathsf{P.F}$. Propositional formulas $\psi$ are formulas over $\mathsf{P.F}$ and $\mathsf{F}$. A DVPL also has an optional set of DVPL names $\overline{\mathsf{V}}$ in its **uses** clause.

The deltas and the base program may contain variant references of the form $\mathsf{P}$ (where $\mathsf{P}$ is one of the parameters) or $\mathsf{V}'$ (where $\mathsf{V}'$ is either the $\mathsf{V}$ itself, or one one of the DVPLs listed in the **uses** clause). In the glue program, variant references

```
module SLine_SMd;
export *;
interface SMd { ... }

module SLine_Pre_Form_BMd;
import Sig from SLine_SMd;
export *;
class Signal implements SLine_SMd.Sig {...}

module SLine_Main_Form_BMd;
import Sig from SLine_SMd;
export *;
class Signal implements SLine_SMd.Sig {...}


module main;
import Signal from SLine_Pre_Form_BMd;
import Signal from SLine_Main_Form_BMd;
import Sig    from SLine_SMd;
class Main {
  Unit main() {
    SLine_SMd.Sig s1 = new SLine_Pre_Form_BMd.Signal();
    SLine_SMd.Sig s2 = new SLine_Main_Form_BMd.Signal();
    s1.connect(s2);
    SLine_SMd.Sig s3 = new SLine_Pre_Form_BMd.Signal();
    SLine_SMd.Sig s4 = new SLine_Main_Form_BMd.Signal();
    s3.connect(s4);
  }
}
```

**Fig. 5** FAM program obtained by flattening the glue program in Fig. 4 under the assumption that no auxiliary classes or interfaces are added by the activated deltas (see the explanation in Sect. 3.3).

to DVPLs have the form $\mathsf{V}[\overline{\mathsf{F}}](\mathsf{VR})$: in addition to features, they may depend on variants of other product lines declared as parameters. The variants listed in the parameters VR must select products of a matching product line in accordance with the DVPL's declaration. All other clauses are defined as in Sects. 2, 3.

$$\mathsf{Dvpl} ::= \textbf{productline}\ \mathsf{V}(\overline{\mathsf{V}\ \mathsf{P}});\ [\textbf{uses}\ \overline{\mathsf{V}};]\ \textbf{features}\ \overline{\mathsf{F}}\ \textbf{with}\ \psi;\quad \mathsf{Prgm}\ \textbf{unique}\{\overline{\mathsf{Mod}}\}\ \overline{\Delta}\ \mathsf{DConfig} \qquad \text{DVPL}$$

$$\mathsf{VR} ::= \mathsf{V} \mid \mathsf{V}[\overline{\mathsf{F}}](\mathsf{VR}) \mid \mathsf{P} \qquad \text{Variant References}$$

**Fig. 6** Syntax of Featherweight Dependent Delta ABS with Modules.

A DVPL supports two kinds of dependencies:

1. It may refer to a variant associated with a parameter P by a prefix of the form P.M, where M is a module name.
2. It may use the **unique** part of other DVPLs: in a DVPL V, any reference to a **unique** class C or interface I from outside must be done with an extended reference of the form V′.C or V′.I. The referenced DVPL V′ (when different from V itself) must be listed in the **uses** clause of V.

All names occurring in the parameters declared by a DVPL are implicitly added to its **uses** clause.

*Example 4.* The following model uses the `Sig` interface of the VPL `SLine` in Fig. 4. The DVPL `BLine` has a dependency on the **unique** part of `SLine`, declared via **uses** `SLine`. As the interface `Sig` is from the **unique** part of the `SLine` VPL, it is unnecessary to refer to any *variant* of `SLine`. Therefore, `BLine` has no parameters.

```
productline BLine;
uses SLine;
unique {
 module ExampleMd;
 interface ExampleI {
   addSignal(SLine.SMd.Sig sig);
}
...
```

Variant generation works bottom-up: variants of DVPLs without parameters are generated first. Variants of other DVPLs are generated by instantiating their parameters with variant selections, once these have been reduced to module references. We provide a more detailed description in Sect. 4.3.

### *4.2 A DVPL for Railway Stations*

Consider the DVPL in Fig. 7 which models a train station with two pre/main signal pairs. The signals within a pair must be implemented with the same technology, i.e. they must be both light signals or both form signals. The feature model ensures this as follows: Parameter `sl1` is constrained to be a pre signal by `sl1.Pre`, similarly `sl2` must be a main signal. The first equivalence ensures that both feature the same technology. Finally, the features of the variants referenced in the parameters are consistently connected to the features of `BlockLine`. There is no **uses** dependency to `SLine`, as it occurs in the parameters.

The attempt to pass two main signal variants or a light pre signal and form main signal to the parameters of `BlockLine` causes variant generation to fail. A correct instantiation of `BlockLine`, for example, with light signals is:

```
BlockLine[Light](SLine[Light, Pre](), SLine[Light, Main]())
```

```
productline BlockLine(SLine sl1, SLine sl2);
features Light, Form with sl1.Form ↔ sl2.Form ∧ sl1.Pre ∧ sl2
 .Main ∧
                         Light ↔ sl1.Light ∧ Form ↔ sl1.
 Form;
delta AlwaysDelta;
adds interface BlMd.BlockI { ... }
adds class BlMd.Block implements BlMd.BlockI {
    SLine.SMd.Sig s1 = new sl1.BMd.Signal();
    SLine.SMd.Sig s2 = new sl2.BMd.Signal();
    SLine.SMd.Sig s3 = new sl1.BMd.Signal();
    SLine.SMd.Sig s4 = new sl2.BMd.Signal();
    Unit Block() {
        s1.connect(s2);
        s3.connect(s4);
    }
}
delta AlwaysDelta when True;
```

**Fig. 7** A DVPL modeling a railway block station.

Dependent product lines can declare other dependent product lines as parameters. The DVPL in Fig. 8 models a railway line with two block stations that reference the neighboring signal of each other. It adds a class `Line` in module `LMd` with its block stations and their facing signals as fields. The `BlockI` interface from `BlockLine` is not unique and thus must be referenced in the products `bl1`, `bl2`. Interface `Sig` , however, is referenced unqualified. No parameter of `LineLine` is from `SLine`, therefore, a dependency **uses** `SLine` is supplied.

```
productline LineLine(BlockLine bl1, BlockLine bl2);
uses SLine;
delta AlwaysDelta;
adds class LMd.Line {
  bl1.BlMd.BlockI b1 = new bl1.BlMd.Block();
  bl2.BlMd.BlockI b2 = new bl2.BlMd.Block();
  SLine.SMd.Sig s1 = b1.getRightSignal();
  SLine.SMd.Sig s2 = b2.getLeftSignal();
  Unit Line() {
    b1.connect(s2);
    b2.connect(s1);
  }
}
delta AlwaysDelta when True;
```

**Fig. 8** A DVPL modeling a railway block section.

### 4.3 Glue Program Flattening for FDDAM

Flattening a FDDAM glue program is based on the procedure described in Sect. 3.4 which must be modified and extended as follows:

1. **Reference Selection.** Variant references may occur nested in FDDAM, so a variant reference or product line without parameters must be selected. That reference is either to a non-dependent VPL, or only contains **uses** dependencies.
2. **Post-Processing of a Single Iteration.** After variant generation for a VPL two additional steps are performed:

   a. If the selected VPL is a DVPL with **uses** V dependencies (but without parameters), then appropriate import clauses of the form **import** `*` **from**$mod(\mathsf{V}, \texttt{mod})$ are added to the generated module, for each `mod` in the unique block of V.
   b. Variant references in the base program or deltas of the DVPL are replaced by module references as described in Sect. 3.4. However, this is not possible when the reference to be resolved is a parameter of a DVPL, because a parameter must have the syntactic shape of a variant reference, not that of a module reference. Instead, DVPLs are partially instantiated: A copy is created where the parameter corresponding to the variant reference to be resolved is instantiated. To resolve references in these copies we use an injective function $dep$ which maps pairs of DVPL names and variant selections to fresh DVPL names. This function is used to generate the fresh names of partially instantiated DVPLs.
   For the overall flattening process we also define an auxiliary function *aux* that maps DVPL names to pairs of module names and class/interface names. This is used to add **import** clauses when the final variant is generated, because the **import** clauses must be added for all parameters of the DVPL. For all DVPLs present in the beginning *aux* is set to $\emptyset$.
   If, during post-processing, the variant reference VR to be resolved occurs in the parameter list of a DVPL, then that DVPL is copied and the following actions are performed *on the copy*:
      i. The DPVL's name V is replaced with $dep(\mathsf{V}, \mathsf{VR})$.
      ii. The instantiated parameter P is removed from the parameter list.
      iii. All features of P occurring in the feature model are replaced with `True` or `False`, depending on whether the feature is selected or not.
      iv. Every reference of the form P.`M` in the deltas or base program of the DVPL is replaced with $mod(\mathsf{VR}, \texttt{M})$.

   The auxiliary function is set to:

$$aux(dep(\mathsf{V},\mathsf{VR})) = \{(mod(\mathsf{VR},\texttt{M}),\texttt{C}) \mid \mathsf{P}.\texttt{M}.\texttt{C} \text{ occurs in } dep(\mathsf{V},\mathsf{VR})\} \cup aux(\mathsf{V})$$

   The DVPL variant reference to be resolved is then replaced with $dep(\mathsf{V}, \mathsf{VR})$. Fig. 9 shows a copy of `BlockLine` after the reference to `SLine[Light, Pre]()` has been resolved. Please observe that the parameter `sl1` is gone and the feature constraint has been simplified using `sl1.Pre` and ¬`sl1.Form`. Moreover, *aux*(`BlockLine_SLine_Light_Pre`) = {(`SLine_Light_Pre_BMd`, `Signal`)}.

3. **Final Post-Processing.** For each (`m`, `C`) ∈ *aux*(V), where VPL V has no parameters (is fully initialized), the clause **import** `C` **from** `M;` is added to all modules generated from V. The final processing of the example in Fig. 9 generates the following import clauses:

```
import Signal from SLine_Light_Pre_BMd;
import Signal from SLine_Light_Main_BMd;
```

```
productline BlockLine_SLine_Light_Pre(SLine sl2);
features Light, Form with ¬sl2.Form ∧ sl2.Light ∧ sl1.Main ∧
 Light ∧ ¬Form;
delta AlwaysDelta;
adds interface BlMd.BlockI { ... }
adds class BlMd.Block implements BlMd.BlockI {
  SLine.SMd.Sig s1 = new SLine_Light_Pre_BMd.Signal();
  SLine.SMd.Sig s2 = new sl2.BMd.Signal();
  SLine.SMd.Sig s3 = new SLine_Light_Pre_BMd.Signal();
  SLine.SMd.Sig s4 = new sl2.BMd.Signal();
  Unit Block() {
    s1.connect(s2);
    s3.connect(s4);
  }
}
delta AlwaysDelta when True;
```

**Fig. 9** The DVPL resulting from partial instantiation of the DVPL in Fig. 7.

## 5 Discussion of Design Decisions

In this section we briefly discuss and explain some central design decisions taken, including possible alternatives.

### *5.1 Variability, Commonality and Interoperability*

We use the **unique** block to share elements common to all variants. This goes beyond the idea that product lines model *only* variability. Instead, DVPLs specify *both*, the variable and the common parts of a concept: In addition to modeling variability, DVPLs are a means to structure the overall code. We chose to place aspects of a model that do not vary over products inside a dedicated **unique** block of a DVPL. Two other possible solutions do not require such a block, but have other downsides:

- One alternative is to place common parts into the glue program. However, moving a referenced interface outside of a product line results in a less coherent overall model: The DVPL is now not a single stand-alone unit, but relies on the correct context (namely the one providing the interface).
- Another solution would be to link a DVPL (modeling the variable part of a concept) and a module (modeling the common part) with a new syntactic construct, to make the coupling explicit. Such an external coupling introduces a new concept to the language and is less elegant than coupling variablity and commonality by including and marking common parts in the DVPL.

Variant references in VPLs are similar to dynamic mixin composition in, for example, Scala. The Scala code below creates an object of class `C` and adds the trait/mixin `T`. During compilation, this is replaced by an anonymous class:

```
val o = new C with T
```

Both, VPLs and dynamic mixins, are used for on-the-fly generation of variant concepts. Despite this, both mechanisms differ in scope and aim:

- VPLs are more general, in the sense that they operate on an arbitrarily large conceptual model. Mixins are confined to single classes.
- Mixins are integrated into the type hierarchy, while the code generated by VPLs merely copies part of the type hierarchy and operates on the copy.

### 5.2 Implementing Interoperability

We decided to base the implementation of interoperability among different variants of a product line on ABS modules and on invariant classes/interfaces of the product line (identified by the **unique** keyword).

Modules constitute an appropriate mechanism to encapsulate different variants with overlapping namespaces. As seen above, the module mechanism cleanly separates the identifiers to access different variants while retaining considerable flexibility over what is visible via the import/export mechanism. In the `Sline` product line, for example, it is possible to specify that the `Sig` interface and the `Signal` class are accessible by the variants, but not by possible auxiliary classes such as a `Bulb` class that might be part of the implementation of a light signal.

Modules are a standard concept, available in ABS (and many other languages) that is sufficient to solve the problem of overlapping name spaces and graded visibility, without the need for dedicated special mechanisms.

As an alternative to modules and **unique** model elements it would have been possible to realize interoperability by a dedicated name space concept plus a type system. This would, however, require the introduction of new concepts that arguably are harder to comprehend.

## 6 Related Work

Kästner et al. [18] 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. A major difference to our proposal is their approach to implement variability (to build variants): they use an *annotative approach* (`#ifdef` preprocessor directives), while we use a *transformational approach* (DOP)—see [22, 26] for a classification and survey of different approaches to implement variability.

Schröter et al. [24] advocate investigating mechanisms to support compositional analyses of MPLs for different stages of the development process. In particular, they outline the notion of *syntactical interfaces* to provide a view of reusable programming artifacts, as well as *behavioral interfaces* that build on syntactical interfaces to support formal verification. Schröter et al. [25] propose *feature-context interfaces* aimed at supporting type checking SPLs developed according to the FOP approach which, as pointed out in Sect. 1, is encompassed by DOP (see [21] for a detailed comparison between FOP and DOP). A feature-context interface supports type checking of 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. More recently, Schröter et al. [23] proposed a concept of feature model interface (based on the feature model slicing operator introduced by Acher et al. [1]) 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.

Damiani et al. [12] informally outline linguistic constructs to extend DOP for SPLs of Java programs to implement MPLs. The idea is to define an MPL as an SPL that imports other SPLs. This extension is very flexible, however, it does not enforce any boundary between different SPLs: the artifact base of the importing SPL is interspersed with the artifact bases of the imported SPLs. Thus the proposed constructs are not suitable for compositional analyses. More recently, Damiani et al. [10] extended the notions proposed in [23] from feature models to complete SPLs. They propose, in the context of DOP for SPLs of Java programs, the concepts of SPL Signature (SPLS), Dependent SPL (DSPL), and DSPL-DSPL composition and show how to use these concepts to support compositional type checking of delta-oriented MPL (by relying on existing techniques for type checking DOP SPLs [4, 11, 8]). An SPLS 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 DSPL are intended to be accessible by variants of other DSPLs.

In contrast to feature-context interfaces [25], the concept of SPLS [10] represents a variability-aware API that supports compositional type checking of MPLs.

None of the above mentioned proposals contains a mechanism for interoperation of multiple variants from the same product line in the same application, the main goal of the present paper. The concept of DVPLs over core ABS programs proposed in this paper, formalized in the FDDAM language, is closely related to the notion of DSPL of Java programs by Damiani et al. [10], formalized in IFM$\Delta$J—a calculus for product lines where variants are programs written in IFJ [12] (an imperative version of Featherweight Java [15]). In particular, both approaches support to model an MPL as a set of dependent product lines. The main differences are as follows:

- IFM$\Delta$J uses SPLSs, syntactic interfaces providing variability-aware APIs, to express the dependencies of a product line. In IFM$\Delta$J a DSPL has anonymous parameters described by SPLS names.
  - On one hand, parameters in IFM$\Delta$J are more flexible than parameters in FDDAM, since they can be instantiated by suitable variants of any product line that implements the associated SPLS—in contrast to FDDAM, where each parameter of a DVPL is associated with a specific product line name.
  - On the other hand, parameters in IFM$\Delta$J are less flexible than parameters in DSPL, since in IFM$\Delta$J a DSPL cannot have more than one parameter for each SPLS and different parameters must be instantiated by variants of different product lines—in contrast to FDDAM, where each parameter has a name and it is possible to have different parameters associated to the same product line.
- FDDAM provides **unique** blocks and glue programs to write applications that reference different variants (possibly from the same product line) and make them interoperate. In contrast, IFM$\Delta$J does not provide any mechanisms to write applications that reference different variants from the same product line.

FDDAM is more suited for our model of railway operations in `FormbaR`: the product line for a parameter is always known beforehand. As shown in the examples, interoperable variants occur naturally in this domain.

## 7 Conclusion

We proposed the concept of dependent variant-interoperable software product lines (DVPL). It provides novel linguistic mechanisms that support interoperability among different variants of one product line and enables describing an MPL by a set of DVPLs and a glue program that may contain references to different variants of the DVPLs. We have illustrated our proposal as an extension of a foundational language for ABS, a modelling language that supports delta-oriented SPLs, and outlined its application to a case study from the `FormbaR` project performed for Deutsche Bahn AG.

In future work we would like to fully formalize our proposal and develop a compositional type-checking analysis for MPLs described according to our proposal. The starting point for developing the analysis is represented by the work of Damiani et al. [10] (see Sect. 6). Furthermore, we plan to implement the proposal for full ABS.

## Acknowledgments

This work is supported by `FormbaR`, part of AG Signalling/DB Raillab (formbar.raillab.de); EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; and ICT COST Action IC1402 ARVI (www.cost-arvi.eu).

## 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.
2. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
4. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
5. P. Clements and L. Northrop. *Software Product Lines: Practices & Patterns*. Addison Wesley Longman, 2001.
6. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. A unified and formal programming model for deltas and traits. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2017.
7. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. Interoperability of software product line variants, 2018. To appear in SPLC'18.
8. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2016.
9. F. Damiani, M. Lienhardt, R. Muschevici, and I. Schaefer. An extension of the ABS toolchain with a mechanism for type checking spls. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, volume 10510 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2017.
10. F. Damiani, M. Lienhardt, and L. Paolini. A formal model for multi spls. In *FSEN*, volume 10522 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2017.
11. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 2012.

12. 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.
13. R. Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In M. Bonsangue, F. de Boer, E. Giachino, and R. Hähnle, editors, *Intl. School on Formal Models for Components and Objects: Post Proceedings*, volume 7866 of *LNCS*, pages 1–37. Springer, 2013.
14. 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.
15. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. 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 - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, pages 142–164, 2010.
17. E. Kamburjan and R. Hähnle. Uniform modeling of railway operations. In *FTSCS*, volume 694 of *Communications in Computer and Information Science*, pages 55–71, 2016.
18. 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.
19. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, Berlin, Germany, 2005.
20. 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.
21. 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.
22. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
23. 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.
24. 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.
25. 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.
26. 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.