Variability Modules

Ferruccio Damiani[1], Reiner Hähnle[2], Eduard Kamburjan[2,4], Michael Lienhardt[3], and Luca Paolini[1]

[1] University of Torino, Torino, Italy
{ferruccio.damiani, luca.paolini}@unito.it
[2] Technische Universität Darmstadt, Darmstadt, Germany
haehnle@cs.tu-darmstadt.de
[3] ONERA, Palaiseau, France
michael.lienhardt@onera.fr
[4] University of Oslo, Oslo, Norway
eduard@ifi.uio.no

Abstract. A Software Product Line (SPL) is a family of similar programs (called variants) generated from a common artifact base. A Multi SPL (MPL) is a set of interdependent SPLs (i.e., such that an SPL's variant can depend on variants from other SPLs). MPLs are challenging to model and implement efficiently, especially when different variants of the same SPL must coexist and interoperate. We address this challenge by introducing variablility modules (VMs), a new language construct. A VM represents both a module and an SPL of standard (variability-free) possibly interdependent modules. Generating a variant of a VM triggers the generation of all variants required to fulfill its dependencies. Then, a set of interdependent VMs represents an MPL that can be compiled into a set of standard modules. We illustrate VMs by an example from an industrial modeling scenario, formalize them in a core calculus, and provide a VM implementation for the active object language ABS.

1 Introduction

The modeling of variability aspects of complex systems poses challenges currently not adequately met by standard approaches to software product line engineering (SPLE) [6,27]. A first modeling challenge is the situation when more than one product line is involved and these product lines depend on each other. Such sets of related and interdependent product lines are known as multi product lines (MPL) [17]. A second modeling challenge is the situation when different product variants from the same product line need to co-exist in the same context and must be interoperable.

In Sect. 2 we exemplify both of these situations in the context of an industrial case study from the literature [20,21], performed for Deutsche Bahn Netz AG, where: (i) several interdependent product lines for networks, signals, switches, etc., occur; and (ii) for example, mechanic and electric rail switches are different variants of the same product line, and some train stations include both. Overall,

multi product lines give rise to the quest for mechanisms for hiding implementation details, reducing dependencies, controlling access to elements, etc. [17].

Challenges similar to those mentioned above were formulated in the SPLC 2018 challenge track [8] and resulted in two proposals. The first [34] is based on an UML-style model, the second [9] is based on the variability-aware active object language ABS [16,19]. The main drawback of either is their relative complexity, which makes these solutions unlikely to be adopted by practitioners. At the same time, supporting interoperability and encapsulation for multiple product variants from multiple product lines remains an important issue to be dealt with in a satisfactory manner. In this paper we propose a novel solution.

The central idea is to take the standard concept of a module, used to structure large software systems since the 1970s, as a baseline. Software modules are implemented in many programming and modeling languages, including ABS, Ada, Haskell, Java, Scala, to name just a few. Because modules are intended to support interoperability and encapsulation, no further *ad hoc* concepts are needed for this purpose. We merely add variability to modules, rendering each module a product line of standard, variability-free modules. We call the resulting language concept *variability module* (VM).

The main advantage of VMs is their conceptual simplicity: as a straightforward extension of standard software modules, they are rather intuitive to use for anyone familiar with modules and with software product lines. Each VM is both a module and a product line of modules. This reduction of concepts not only drastically simplifies syntax, but reduces the cognitive burden on the modeler. We substantiate this claim: in Sect. 2 we illustrate the railways MPL case study in terms of VMs without the need to introduce any formal concepts. Only in Sect. 3 we introduce the syntax of VMs formally as an extension of the standard module system of the ABS language.

We stress that VMs can be added to any class-based programming language that has modules. Moreover, while variability support is a major implementation issue, the choice of the *specific* approach used to implement variability [1] is orthogonal to the concept of VM. The VM incarnation given in this paper is based on the ABS language [16,19], which supports *delta-oriented programming* (DOP) [1, Sect. 6.6.1], [29] to implement variability. We chose ABS mainly because it features a native implementation of DOP that was successfully used in industrial case studies [21,25,38].

The formal underpinnings of VMs are covered in Sects. 3–5. In Sect. 3 we declare the VM syntax and spell out consistency requirements. Sect. 4 formalizes a statically checkable property of VMs: the *principle of encapsulated variability*, which ensures that any dependency among VMs can be reduced to dependencies among standard variability-free modules. In Sect. 5 we define variant generation in terms of a "flattening" semantics: the variants requested from an SPL represented by a VM, together with necessary variants of other VMs it depends upon, are generated by translating each VM into a set of variability-free, standard modules (one for per variant). This results in a variability-free program

with suitably disambiguated identifiers and is sufficient to define the semantics of VMs precisely, to compile and to run them.

Sect. 6 describes how the VM concept is integrated into the existing ABS tool chain and evaluates it by means of case studies. The main point is that, as long as one has control over the parser and abstract syntax tree, it is relatively straightforward to realize the flattening algorithm of Sect. 5. In Sect. 7 we discuss the main decisions taken during VM design. We discuss related work in Sect. 8 and conclude in Sect. 9 by outlining ongoing work.

2 Overview and Motivating Example

We illustrate VMs with an example based on an industrial case study from railway engineering [21]. The scenario contains signals, switches, interlocking systems that use multiple variants of signals and switches, and a railway station that uses multiple variants of interlocking systems.

Fig. 1 shows the feature models for switches and signals.[5] A signal is either a light signal, using bulbs and colors to indicate the signal aspect or a form signal that uses mechanically moved shapes. All variants of signal have the interface to the interlocking system and basic functionality, such as aspect change, in common (e.g., signal can always signal the aspects "Halt" and "Go"). If multiple outgoing tracks are possible, a signal may also indicate the direction the train is going—so there are 4 variants of signal. Variability is implemented, e.g., by the presence of an additional class `Bulb` in the variant for light signals and by the fact that method `setToHalt` (which changes the shown aspect to "Halt") is different for form signals and for light signals (as these must communicate with their `Bulb` instances).

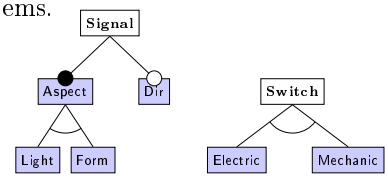

Fig. 1. Feature Models for Signals and Switches.

Signals are modeled by the VM `Signals` in Fig. 2. It starts with the keyword `module`, followed by the module name, by the list of exported module elements, and by the feature list constrained with a propositional formula describing the products. The module header is terminated by a list of configuration definitions (here just one) and by a list of product definitions (here just one), where each definition gives a name to a set of features. Next there are the definition of interface `ISig` and the definition of class `CSig` implementing `ISig`. By default, class and interface definitions can be modified/removed by deltas to obtain different product variants. Class/interface definitions annotated by the keyword `unique` must be the same in all product variants. They enable interoperability between different product variants of the same VM. These definitions are followed by the deltas

[5] Feature models [1] specify software variants in terms of features. A feature is a name representing some functionality, a set of features is called a configuration, and each variant is identified by a valid configuration (called a product, for short). Equivalent representations of feature models have been proposed in the literature, like feature diagrams (see Fig. 1) and propositional formulas (see line 3 in Figs. 2 and 3).

```
// MODULE HEADER
module Signals;   export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
configuration KSig = {Dir};
product LSig = {Light};
// CORE PART
unique interface ISig { Bool eqAspect(ISig); Unit setToHalt(); }
class CSig implements ISig { }
// DELTA PART
delta LDelta; adds class CBulb { };
              modifies class CSig { Unit addBulb() { new CBulb();} };
delta FDelta; modifies class CSig { };
delta DDelta; adds interface IDir { };
              adds class CDir implements IDir{ };
              modifies class CSig { adds IDir getDirection() { } };
delta LDelta when Light; delta FDelta when Form; delta DDelta when Dir;
```

Fig. 2. An SPL of signals as a VM.

```
// MODULE HEADER
module Switches;   export ITrack, CTrack, CSwitch, ISwitch;
features Electric, Mechanic with Electric <-> !Mechanic;
// CORE PART
unique interface ISwitch {  }
class CSwitch implements ISwitch {  }
unique interface ITrack { ISwitch appendSwitch(); }
class CTrack implements ITrack {
   ISwitch appendSwitch() { ISwitch sw = new CSwitch(); return sw; }
}
// DELTA PART
delta EDelta; modifies class CSwitch { };
delta MDelta; modifies class CSwitch { adds Bool isMechanic() {return True;}};
delta EDelta when Electric; delta MDelta when Mechanic;
```

Fig. 3. An SPL of tracks and switches as a VM.

that describe the implementation of different variants and their application conditions. Classes and interfaces added by deltas can be modified/removed by other deltas. The delta `LDelta`, triggered by feature Light, adds a class `Bulb` and modifies the class `CSig` to reference the class `Bulb`. Deltas `FDelta` and `DDelta` implement features Form and Dir, respectively.

Switches and tracks are modeled by the VM in Fig. 3. It is structurally similar to `Signals`. A switch is either electric (controlled from the interlocking system) or mechanic (controlled locally by a lever). Class `CTrack` contains a reference to class `CSwitch`, which is not declared `unique`. So, even though class `CTrack` is not modified/removed by any delta, its declaration cannot be annotated with `unique`.

Interlocking systems are modeled by the VM in Fig. 4. An interlocking system manages switches and signals that lie on tracks, it imports all the exported elements (feature names are implicitly exported/imported) of the VMs `Signals` and `Switches`. The VM `InterlockingSys` has four variants, modeled by two optional features. Line 6 contains a product definition that gives name `PSwitch` to a product of the VM `Switch`. It is called an *open* product definition, because it depends on the selected product of the VM `InterlockingSys` itself: if feature `Modern` is selected `PSwitch` specifies an electric switch, otherwise it specifies a mechanic switch (product clauses are evaluated in order until a valid one is found). Line 7 contains an

```
1  // MODULE HEADER
2  module InterlockingSys;
3  export *;
4  import * from Signals; import * from Switches;
5  features Modern, DirOut with True;
6  product PSwitch for Switches = { Modern => {Electric}, !Modern => {Mechanic} }
7  product PSignal for Signals  =
8     { DirOut &&  Modern => {Light,Dir}, Modern => {Light},
9       DirOut && !Modern => {Form,Dir}, !Modern => {Form}  }
10 // CORE PART
11 unique interface IILS {  }
12 class CILS {
13   Bool testSig() {
14     ISwitch swNormal = new CSwitch() with {Electric};
15     ITrack track     = new CTrack() with {Mechanic};
16     ISwitch swNew    = track.appendSwitch();
17     ISig sigNormal   = new CSig() with LSig;
18     ISig sigShunt    = new CSig() with {Form};
19     return sigNormal.eqAspect(sigShunt);
20   }
21   ISwitch createSwitch() { return new CSwitch() with PSwitch;}
22   ISignal createOutSignal() { return new CSignal() with PSignal;}
23   ISignal createInSignal() { return new CSignal() with PSignal - {Direction};}
24 }
```

Fig. 4. An SPL of interlocking systems as a VM.

open product definition for the VM `Signal`. It is worth observing that open product definitions enable implementing different variants of the VM `InterlockingSys` without using deltas. Method `testSig` of class `CILS` instantiates classes from two different product variants of `Switches` and from two different product variants of `Signals`. All references to non-`unique` imported classes/interfaces specify a product, by using a `with` clause. In a `with` clause, the product can be specified by explicitly listing its features, by using one of the defined product names, or (more generally) by a set-theoretic expression. For example, `track` is taken from product `{Mechanic}` of module `Switches`, while `sigNormal` uses the product name `LSig` imported from `Signals`. In line 16 a switch is added to a track element—since `track` contains a reference to an instance of the mechanic variant of class `CTrack`, `appendSwitch()` will add a mechanic switch. All signal variants of the VM `Signals` share the same definition of the unique `ISig` interface, thus making it accessible to anyone that imports it from `Signals`. On the other hand, the `CBulb` class is only used inside the VM `Signals`. Different product variants are fully interoperable, as witnessed by the expression in line 19.

A VM that does not contain a feature model (and, therefore, no configuration definitions, no open product definitions and no deltas) is called a *variability-free module* (VFM). All classes of a VFM are considered unique (there is no need to write the `unique` keyword). Each program must have exactly one *main module*, that is, is a VMF containing an implicit class providing an initialization method defined by using the keyword `init`. The whole railway station is modeled by a main module, given in Fig. 5, together with the VM `Signals` in Fig. 2, the VM `Switches` in Fig. 3, and the VM `InterlockingSys` in Fig.s 4. It represents a MPL—we call it the railway station MPL.

```
module RailwayStation;
import * from InterlockingSys;
init {
  IILS ils1 = new CILS() with { DirOut };
  IILS ils2 = new CILS() with { Modern };
}
```

Fig. 5. Railway station as a main module.

Prg	::= $\overline{\text{Mdl}}$	**Program**
Mdl	::= MdlH MdlC MdlD	(Variability) Module
MdlH	::= module M; [export tC;] $\overline{\text{import tC from M;}}$	**Module Header**
↪	[features $\overline{\text{F}}$ with Φ; $\overline{\text{KD}}$] $\overline{\text{PD}}$	
tC	::= tN $\overline{\text{tN}}$ | *	Trade Clause
tN	::= C | I | K | P	Traded names
KD	::= configuration K = KE	Configuration Declaration
KE	::= K | P | {$\overline{\text{F}}$} | KE + KE | KE * KE | KE − KE	Configuration Expression
PD	::= product P [for M] = KE | { PC $\overline{\text{,PC}}$ }	Product Declaration
PC	::= Φ => KE	Pattern Clause
MdlC	::= $\overline{\text{Defn}}$ [init { $\overline{\text{S}}$ }]	**Module Core Part**
Defn	::= [unique] ID | [unique] CD	Interface/Class Definition
ID	::= interface I [extends IR $\overline{\text{IR}}$] { $\overline{\text{MH}}$ }	***Interface Definition***
IR	::= I [with KE] | M.I [with KE]	Interface Reference
MH	::= T m ($\overline{\text{T x}}$)	Method Header
T	::= IR | Unit | Int | ...	Type
CD	::= class C [implements IR $\overline{\text{IR}}$] { $\overline{\text{FD}}$ $\overline{\text{MD}}$ }	***Class Definition***
FD	::= T f;	Field Definition
MD	::= MH { $\overline{\text{S}}$ return E;}	Method Definition
S	::= [T] v = E; | E.f = E; | ...	Statement
E	::= x | E.f | E.m($\overline{\text{E}}$) | new CR($\overline{\text{E}}$) [with KE] | ...	Expression
CR	::= C | M.C	Class Reference
MdlD	::= $\overline{\text{Dlt}}$ CK	**Module Delta Part**
Dlt	::= delta D; $\overline{\text{CO}}$ $\overline{\text{IO}}$	***Delta***
CO	::= adds CD | removes class C	Class Operation
↪	| modifies class C [adds IR $\overline{\text{IR}}$] [removes IR $\overline{\text{IR}}$] { $\overline{\text{AO}}$ }	
AO	::= adds AD | removes HD | modifies MD	Attribute Operation
AD	::= FD | MD	Attribute Declaration
HD	::= FD | MH	Header Declaration
IO	::= adds ID | removes interface I	Interface Operation
↪	| modifies interface I [adds IR $\overline{\text{IR}}$] [removes IR $\overline{\text{IR}}$] { $\overline{\text{SO}}$ }	
SO	::= adds MH | removes MH	Signature Operation
CK	::= $\overline{\text{DAC}}$ $\overline{\text{DAO}}$	***Configuration Knowledge***
DAC	::= delta D when Φ;	Delta Activation Condition
DAO	::= D $\overline{\text{D}}$ < D $\overline{\text{D}}$ $\overline{\text{< D D}}$;	Delta Application Order

Fig. 6. ABS-VM abridged syntax.

3 Variability Module Syntax

The abridged syntax of *ABS with VMs* (ABS-VM, for short) is given in Fig. 6. It defines the OO fragment of ABS [16,19], extended with VM concepts. A program is a sequence of VMs—as usual, $\overline{X}$ denotes a possibly empty finite sequence of elements X. A VM consists of a header (`MdlH`), a core part (`MdlC`), and a delta part (`MdlD`). A VM header comprises the keyword `module` followed by the name of the VM, by some (possibly none) `import` and `export` clauses (listing the class/interface/configuration/products names that are exported or imported by the VM, respectively), by the optional definition of a feature model (where $\overline{\texttt{F}}$ are the features and Φ is a propositional formula over features), by a list of configuration definitions and by a list of product definitions. A configuration expression `KE` is a set-theoretic expression over sets of features ($+$, $*$ and $-$ denote union, intersection and difference, respectively). A product definition `PD` is *closed* if it is of the form `product P [for M] = KE`, otherwise it is *open*. The clauses in an open product definition are examined in sequence until the first valid clause is found. The right-hand sides of configuration definitions do not contain product names, and the right-hand sides of closed product definitions do not contain open product names. Recursive configuration/product definitions are forbidden.

Both the module core part and the module delta part may be empty. A module core part comprises a sequence of class and interface definitions $\overline{\texttt{Defn}}$. As an extension to ABS syntax, each of these definitions may be prefixed by the keyword `unique`. Each use of a class, interface or product name imported from another module may be prefixed by the name of the module—the name of the module *must* be used if there are ambiguities (e.g., when an interface with name `I` is imported from two modules). Moreover, each use of a non-unique class or interface imported from another module must be followed by a `with`-clause, specifying (by means of a configuration expression) the variant of the VM it is taken from. From now on, we consider only ABS-VM programs containing one main module (see the last paragraph of Sect. 2) and such that any other VM does not contain the keyword `init`. Observe that a VFM (see the last paragraph of Sect. 2) without product definition (a VFM may contain a closed product definition of the form `product P for` $\cdots$) and no occurrences of the `with` keyword is a variability-free ABS module.

A module delta part comprises a sequence of delta definitions $\overline{\texttt{Dlt}}$ followed by configuration knowledge `CK`. Each delta specifies a number of changes to the module core part. A delta comprises the keyword `delta` followed by the name of the delta, by a sequence of class operations $\overline{\texttt{CO}}$ and by a sequence of interface operations $\overline{\texttt{IO}}$. An *interface operation* can add or remove an interface definition, or modify it by adding/removing names to the list of the extended interfaces or by adding/removing method headers. A *class operation* can add or remove a class definition, or modify it by adding/removing names to the list of the implemented interfaces, by adding/removing fields or by adding/removing/modifying methods. Modifying a method means to replace its body with a new body. The new body may call the reserved method name `original`, which during delta application is bound to the previous implementation of the method.

Configuration knowledge CK provides a mapping from products to variants by describing the connection between deltas and features: it specifies an activation condition Φ (a propositional formula over features) for each delta D by means of a DAC clause; and it specifies an application ordering between deltas by means of a sequence of DAO clauses. Each DAO clause specifies a partial order over the set of deltas in terms of a total order on disjoint subsets of delta names—a DAO clause allows developers to express (as a partial order) dependencies between the deltas (which are usually semantic "requires" relations [3]). The overall delta application order is the union of these partial orders—the compiler checks that the resulting relation R represents a specification that is *consistent* (i.e., R is a partial order) and *unambiguous* (i.e., all the total delta application orders that respect R generate the same variant for each product). Techniques for checking that R is unambiguous are described in the literature [3,23].

The following definition of normal form formalizes a minimum consistency requirement on ABS-VM programs.

Definition 1 (ABS-VM Normal Form). *An ABS-VM program* Prg *is in* normal form *if all its VMs* M *satisfy the following conditions:*

1. *All class references* CR *and interface references* IR *occurring in* M *are qualified, that is of the form* M'.N *for some module name* M' *and class/interface name* N*, and if* M'≠M *then* M *contains the import clause* `import N from M'`.
2. *If* M *contains a clause* `import tC from M'` *then* M'≠M *and all traded names in* tC *occur in the export clause of* M'*. No open product names are exported.*
3. *Let* M *contain a definition*
   (a) `configuration K = KE`. *Then: (i) all feature names occurring in* KE *are features of* M *and all the configuration names occurring in* KE *have been already defined in* M*; and (ii) no product name occurs in* KE*.*
   (b) `product P = KE` *then: (i) condition (a).(i) above holds; (ii) all the product names occurring in* KE *have been already defined in* M *and are closed; and (iii)* KE *denotes a product of* M*.*
   (c) `product P for M' = KE`. *Then: (i)* M ≠ M'*; (ii) all feature names occurring in* KE *are features of* M' *and all configuration/product names occurring in* KE *are imported from* M'*; and (iii)* KE *denotes a product of* M'*.*
   (d) `product P for M' = {`Φ_1`=>` KE_1`,...,`Φ_n`=>` KE_n`}` $(n \geq 1)$. *Then: (i)* M ≠ M'*; (ii) for all* j $(1 \leq j \leq n)$*, all configuration/product names occurring in* KE_j *are imported from* M'*; (iii) for all* j $(1 \leq j \leq n)$*, all feature names occurring in* Φ_j *are features of* M*, all feature names occurring in* KE_j *are features of* M'*, at least one product of* M *satisfies* $(\bigwedge_{1 \leq i < j} \neg \Phi_i) \wedge \Phi_j$*, and* KE_j *denotes a product of* M'*; and (iv) all products of* M *satisfy* $\bigvee_{1 \leq i \leq n} \Phi_i$*.*
   (e) `product P = {`Φ_1`=>` KE_1`,...,`Φ_n`=>` KE_n`}` $(n \geq 1)$. *Then: (i) for all* j $(1 \leq j \leq n)$*, all configuration/product names occurring in* KE_j *have been already defined in* M*; (ii) the condition obtained from condition (d).(iii) above by replacing* M' *by* M *holds; and (iii) condition (d).(iv) above holds.*
4. *If* M *contains an occurrence of* `new M'.C(...) with KE` *or* `M'.I with KE`*, then: (i) all feature names occurring in* KE *are features of* M'*; (ii) if* M'=M *then all configuration/product names occurring in* KE *are defined in* M*; (iii) if* M'≠M *then*

all configuration/product names occurring in `KE` *are either imported from* `M'` *or defined in* `M` *by a declaration of the form* `product P for M' = ...`*; and (iv)* `KE` *denotes a product of* `M'`*.*

Checking whether a program can be transformed into normal form (and, if so, doing it) is straightforward—for each VM `M`: conditions 1, 3.(a)–(b), 3.(e) and (when `M'`=`M`) condition 4 can be ensured by inspecting only `M`; condition 2 and (when `M'`≠`M`) condition 4 can be ensured by inspecting only `M` and the header of the modules `M'` mentioned in the import clause of `M`. Conditions 3.(b).(iii), 3.(c).(iii), 3.(d).(iii)–(iv) and 4.(iv) can be checked with a SAT solver. Programs that cannot be transformed into normal form are rejected by the compiler.

4 Principle of Encapsulated Variability

According to the VM semantics, unique class/interface declarations in a VM `M` are shared by all variants of `M`. The *principle of encapsulated variability* (PEV) prescribes that each VM can depend on other VMs only by using classes or interfaces that are either unique or that belong to a *specific* variant. If a VM program adheres to the PEV, then flattening (defined in Sect. 5)—which removes variability and generates those variants required by the dependencies—can resolve all dependencies among VMs to dependencies among (standard) ABS modules.

In this section we only consider programs in normal form (Def. 1). To formalize and to implement automated checking of PEV adherence we introduce the notion of dependency and the functions CORE, UNIQUE and BASE.

Definition 2 (Dependency). *A VM* `M'` depends on *a VM* `M` *if* `M'` *contains an occurrence of* `M.N` *where* `N` *is a class/interface name. An occurrence of* `M.I with KE` *or* `new M.C(...) with KE` *is called* `with`-dependency *(on* `M.I` *or* `M.C`*, respectively), while an occurrence of* `M.I` *or* `new M.C(...)` *(i.e. not followed by a* `with`*) is called* `with`-free-dependency *(on* `M.I` *or* `M.C`*, respectively). An occurrence of* `M.I with KE` *or* `M.C(...) with KE` *is called* `with`-open-dependency *if* `KE` *contains an occurrence of an open product name* `P`*, it is called* `with`-closed-dependency *else.*

Definition 3 (Functions CORE, UNIQUE, BASE). *Given a program* `Prg`*, for all VMs of* `M` *of* `Prg`*:* CORE*(*`Prg`*,*`M`*) is the set of qualified names* `M.N` *of all interfaces/classes* `N` *whose definition occurs in the core part of* `M`*;* UNIQUE*(*`Prg`*,*`M`*)* ⊆ CORE*(*`Prg`*,*`M`*) contains those class/interface names whose declaration is annotated with* `unique`*;* BASE*(*`Prg`*,*`M`*)* ⊆ CORE*(*`Prg`*,*`M`*) contains those class/interface names that are modified, removed or added by some delta of* `M`*.*

Now we can formalize the PEV as follows:

Definition 4 (Principle of Encapsulated Variability (PEV)). *A program* `Prg` *(in normal form) adheres to PEV, if for all VMs* `M` *of* `Prg`*:*

1. UNIQUE(`Prg`,`M`) ∩ BASE(`Prg`,`M`) = ∅.

2. *For all* `M.N` $\in$ UNIQUE(`Prg`,`M`) *the definition* `Defn` *of* `N` *(in the core part* `MdlC` *of* `M`*) does not contain* `with`*-open-dependencies and, for all* `with`*-free-dependencies on* `M.N'` *occurring in* `Defn`*, it holds that* `M.N'` $\in$ UNIQUE(`Prg`,`M`).
3. *For all* `with`*-free-dependencies on* `M'.N` *occurring in* `M`*: if* `M'` $\neq$ `M` *then* `M'.N` $\in$ UNIQUE(`Prg`,`M'`).

To check whether a program adheres to the PEV is straightforward. Programs that do not adhere to PEV are rejected by the compiler. According to the PEV, VMs can support two types of interaction among variants:

Variant interoperability. Different variants of the *same* VM can coexist and cooperate via unique classes/interfaces. For instance, in the interlocking system MPL of Sect. 2, all *interfaces* are unique and all *classes* are not unique (which is a common pattern). Then, in line 19 of Fig. 4, an instance of class `CSig` in the variant of VM `Signal` for product `{Light}` receives an invocation of method `eqAspect` that (accepts an argument of type `Signal` as formal parameter and) takes as parameter an instance of `CSig` in the variant of `Signal` for product `{Form}`.

Variant interdependence. The code of a variant of a VM $\mathtt{M}_1$ can depend on the code of a variant of a VM $\mathtt{M}_2$ (and possibly vice versa). I.e., the code of $\mathtt{M}_1$ refers to unique classes/interfaces of $\mathtt{M}_2$ (via `with`-free-dependencies) or to classes/interfaces of a specific variant of $\mathtt{M}_2$ (via `with`-dependencies). A special case of variant interdependence is when $\mathtt{M}_1 = \mathtt{M}_2$, i.e., $\mathtt{M}_1$ has a `with`-dependency on a class/interface of $\mathtt{M}_1$ itself. Then in the flattened program a variant of $\mathtt{M}_1$ will contain an occurrence of a class/interface name that is declared in a different variant of $\mathtt{M}_1$.

The following definition and theorem show it is possible to automatically infer the maximal set of class/interface declarations that can be annotated with `unique`.

Definition 5 (Function MaxUNIQUE). *For every program* `Prg`*, for all VM* `M` *of* `Prg`*, let* $S_{\mathtt{M}}$ = CORE(`Prg`,`M`) $\setminus$ BASE(`Prg`,`M`) *and let* $F_{\mathtt{M}} : (2^{S_{\mathtt{M}}}, \subseteq) \to (2^{S_{\mathtt{M}}}, \subseteq)$ *be the non-increasing monotone function such that:* $F_{\mathtt{M}}(X)$ *is the subset of* X *obtained by removing simultaneosly all classes/interfaces* `M.N` *such that the definition of* `N` *in the core part of* `M` *contains a* `with`*-free-dependency on a class/interface* `M.N'` $\notin X$ *or contains a* `with`*-open-dependency. Then* MaxUNIQUE(`Prg`,`M`) *is the set computed by iterating* $F_{\mathtt{M}}(X)$ *on* $S_{\mathtt{M}}$ *until a fixpoint is reached, i.e., the set* $U = F^n_{\mathtt{M}}(S_{\mathtt{M}})$ *such that* $U = F_{\mathtt{M}}(U)$ *for some* $n \geq 0$.

Observe that MaxUNIQUE(`Prg`,`M`) is computed locally on the VM `M` and always terminates (since $F_{\mathtt{M}}$ is non-increasing monotone). Unfortunately, a program `Prg'` such that, for all VM `M` of `Prg'`, UNIQUE(`Prg'`,`M`) = MaxUNIQUE(`Prg'`,`M`) may not adhere to PEV because of point 3 in Definition 4. However (by the following theorem), if such a `Prg'` does not adhere to PEV, then any program obtained from `Prg'` by adding or removing `unique` annotations does not adhere to PEV.

Theorem 1 (Maximal set of `unique` annotations enforcing the PEV). *For all programs* `Prg` *adhering to the PEV: (i) for all VM of* `Prg` `M`*,* UNIQUE(`Prg`,`M`) $\subseteq$

MaxUNIQUE(Prg,M); *(ii) the program* Prg' *obtained by adding* unique *annotations to* Prg *until, for all VM* M, UNIQUE(Prg',M) = MaxUNIQUE(Prg',M), *adheres to PEV.*

Proof. By Def. 5 $F_{\mathtt{M}}$ is a set-theoretic inclusion-preserving map and the power-set $2^{S_{\mathtt{M}}}$ is a complete lattice, so by the Knaster-Tarski theorem [28] there exist smallest and greatest fixpoints of $F_{\mathtt{M}}$. Moreover, the PEV requires (second item of Def. 4) that UNIQUE(Prg,M) is a fixpoint of $F_{\mathtt{M}}$. Now (i) holds, because MaxUNIQUE(Prg,M) is the greatest fixpoint of $F_{\mathtt{M}}$—the proof is as follows: let G be the greatest fixpoint of $F_{\mathtt{M}}$; clearly $G \subseteq S_{\mathtt{M}}$ and (since $F_{\mathtt{M}}$ is non-increasing monotone) $G = F^n_{\mathtt{M}}(G) \subseteq F^n_{\mathtt{M}}(S_{\mathtt{M}})$ for all $n \in \mathbb{N}$; but MaxUNIQUE(Prg,M) is the fixpoint obtained by iterating $F_{\mathtt{M}}$ on $S_{\mathtt{M}}$. And (ii) holds, because Prg' satisfies Def. 4—in particular: the first item holds by definition of $S_{\mathtt{M}}$ and non-increasing monotonicity of $F_{\mathtt{M}}$; the second item is satisfied by any fixpoint of $F_{\mathtt{M}}$; since Prg satisfies the third item, so does Prg'. □

5 Semantics

In this section, we assume (without loss of generality): (i) the considered program Prg is in normal form and adheres to the PEV; (ii) all configuration definitions KD and closed product definitions PD have been resolved in Prg, i.e. all occurrences of their names are removed from export/import clauses of Prg, and all their remaining occurrences in Prg were replaced with their value (a set of features).

In Sect. 5.1 we introduce auxiliary functions for the extraction of relevant information from ABS-VM programs. Then, in Sect. 5.2 we give rewrite rules for transforming an ABS-VM program into a variability-free ABS program.

5.1 Auxiliary Functions

Definition 6 (Lookup Functions). *Given a VM* M *of* Prg *we define the sets:*

- *mdlUnique*(Prg, M) *for the interface/class definitions* $\overline{\texttt{Defn}}$ *in the Module Core Part* MdlC *of* M *annotated with* unique*, and mdlNotUnique*(Prg, M) *for the remaining interface/class definitions.*
- *mdlInit*(Prg, M) *for the init block of* M*, if it exists, or the empty sequence otherwise; also mdlInit*(Prg) *for the name* M *of the VM such that mdlInit*(Prg, M) *is not the empty sequence.*
- *mdlDelta*(Prg, M, π) *where* π *is a product of* M*, for any ordered sequence* $\overline{\texttt{Dlt}}$ *containing exactly those deltas of* M *that are activated by* π*, respecting the order among deltas specified in the configuration knowledge of* M*.*

The meaning of with-free-dependencies or open with-dependencies δ occurring in a VM M is relative to the product π of M being considered. We say a dependency δ is *ground* to mean that it is either with-free or the configuration expression KE in its with-clause is a set of features $\pi = \{\mathtt{F}_1,\ldots,\mathtt{F}_n\}$ $(n \geq 0)$. The following definition introduces notations for extracting the meaning of ground dependencies occurring in the core part of a given VM of a given program.

Definition 7 (Ground Dependency Meaning). *Given a program* Prg*, a VM* M *of* Prg*, a ground dependency* δ *on* M'.N *occurring in* M*, let* xc *be either the symbol* $\bot$ *or a product* π *of* M*. We define:*

$$\Downarrow(\texttt{Prg}, \texttt{M}, \delta, xc) = \begin{cases} (\texttt{M'}, \bot) & \textit{when } \texttt{M'.N} \in \mathsf{UNIQUE}(\texttt{Prg},\texttt{M'}) \\ (\texttt{M'}, xc) & \textit{when } \texttt{M'} = \texttt{M},\ \delta \textit{ is } \texttt{with}\textit{-free and } \texttt{M'.N} \notin \mathsf{UNIQUE}(\texttt{Prg},\texttt{M}) \\ (\texttt{M'}, \pi) & \textit{when } \delta \textit{ is } \texttt{M'.N with } \pi \textit{ and } \texttt{M'.N} \notin \mathsf{UNIQUE}(\texttt{Prg},\texttt{M'}). \end{cases}$$

Let all the dependencies occurring in the core part MdlC *of* M *be ground. Then we define* $\Downarrow(\texttt{Prg}, \texttt{M}, \texttt{MdlC}, xc) = \{\Downarrow(\texttt{Prg}, \texttt{M}, \delta, xc) \mid \delta \textit{ is a dependency in } \texttt{MdlC}\}$.

Flattening a program Prg may require to generate more than one variant for each of its VMs. The flattening process generates new names for the generated variability-free ABS modules implementing the required variants, and translates the dependencies occurring in Prg to suitable dependencies using the generated names. The following definition introduces notations for the names of the generated modules and the translation of with- and with-free-dependencies into the corresponding dependencies among non-variable ABS modules.

Definition 8 (New Module Names, Dependency Translation). *Given a program* Prg*, a VM* M *of* Prg*, let* xc *be either* $\bot$ *or a product* π *of* M*. We denote by* $\Uparrow(\texttt{M}, xc)$ *the name of the module that implements the unique part of the variants of* M *if* $xc = \bot$*, or the name of the module that implements the non-unique part of the variant of* M *for product* π*. Moreover, given a ground dependency* δ *on* M'.N*, we define:*

$$\Uparrow(\texttt{Prg}, \texttt{M}, \delta, xc) = \begin{cases} \Uparrow(\texttt{M'}, \bot).\texttt{N} & \textit{when } \texttt{M'.N} \in \mathsf{UNIQUE}(\texttt{Prg},\texttt{M'}) \\ \Uparrow(\texttt{M'}, xc).\texttt{N} & \textit{when } \texttt{M'} = \texttt{M},\ \delta \textit{ is } \texttt{with}\textit{-free and } \texttt{M'.N} \notin \mathsf{UNIQUE}(\texttt{Prg},\texttt{M}) \\ \Uparrow(\texttt{M'}, \pi).\texttt{N} & \textit{when } \delta \textit{ is } \texttt{M'.N with } \pi \textit{ and } \texttt{M'.N} \notin \mathsf{UNIQUE}(\texttt{Prg},\texttt{M'}). \end{cases}$$

Let all the dependencies occurring in the core part MdlC *of* M *be ground. Then we define* $\Uparrow(\texttt{Prg}, \texttt{M}, \texttt{MdlC}, xc)$ *as the VM core* MdlC' *obtained from* MdlC *by replacing each dependency* δ *occurring in it with* $\Uparrow(\texttt{Prg}, \texttt{M}, \delta, xc)$.

5.2 Flattening

The following definition formalizes the application of an ordered sequence of deltas $\overline{\texttt{Dlt}}$ (the deltas activated by a product π of a VM M) to to a sequence $\overline{\texttt{Defn}}$ of interface/class definitions (the non-unique class/interface definitions in the module core part MdlC of M) as described for ABS in the literature—e.g., for the Featherweight Delta ABS (FDABS) calculus [7].

Definition 9 (Delta Application). *Given a sequence of declarations* $\overline{\texttt{Defn}}$ *and an ordered sequence of deltas* $\overline{\texttt{Dlt}}$*, we write* $(\overline{\texttt{Dlt}}, \overline{\texttt{Defn}}) \rightarrow^* \overline{\texttt{Defn'}}$ *to mean that* $\overline{\texttt{Defn'}}$ *is the outcome of the procedure described in Appendix A.1.*

For all VMs M of Prg, if the products π of M are given, then the right-hand side of each open product definition `product P = ···` or `product P for M' = ···` in M can

be evaluated to a product. This defines a mapping, denoted with $\mathsf{genP}(\texttt{Prg}, \texttt{M}, \pi)$, from open product names to products. Moreover, given a sequence of interface/-class definitions $\overline{\texttt{Defn}}$ and a mapping σ from open product names (at least the open product names occurring in $\overline{\texttt{Defn}}$) to products, we denote with $\sigma(\overline{\texttt{Defn}})$ the definitions obtained from $\overline{\texttt{Defn}}$ by replacing each occurrence of an open product name $\texttt{P}$ with $\sigma(\texttt{P})$.

Definition 10 (Grounding `with`-clauses). *Given a Module Core Part* $\texttt{MdlC}$ *that does not contain occurrences of product names, we write* $\texttt{MdlC} \rightsquigarrow^* \texttt{MdlC'}$ *to mean that the module core part* $\texttt{MdlC'}$ *has been obtained from* $\texttt{MdlC}$ *by replacing the right-hand side of each* `with`*-clause (which is a set-theoretic expression over sets of features) by the corresponding set of features (which, since the overall program is assumed in normal-form, is a product).*

Now we are ready to give the definition of the rules that flatten a VM:

Definition 11 (Flattening a VM). *Let* $\texttt{M}$ *be the name of a VM of* $\texttt{Prg}$*, let* xc *be either the symbol* $\bot$ *or a product* π *of* $\texttt{M}$*. The following rules define a judgment of the form* $\texttt{M} \xrightarrow{\texttt{Prg}, xc} D, \texttt{Mdl}$ *where: (i)* $\texttt{Mdl}$ *is the code of a variability-free ABS module named* $\Uparrow(\texttt{M}, xc)$*, which, for the case* $xc = \bot$ *implements the unique part of the variants of* $\texttt{M}$*, for the case* $xc = \pi$ *it implements the non-unique part of the variant of* $\texttt{M}$ *for product* π*. Moreover,* D *is a set that identifies the variability-free ABS modules that* $\Uparrow(\texttt{M}, xc)$ *depends upon.*

$$\frac{\mathsf{mdlUnique}(\texttt{Prg}, \texttt{M}) = \overline{\texttt{Defn}} \qquad \overline{\texttt{Defn}}\ \mathsf{mdlInit}(\texttt{Prg}, \texttt{M}) \rightsquigarrow^* \texttt{MdlC} \qquad \Downarrow(\texttt{Prg}, \texttt{M}, \texttt{MdlC}, \bot) = D = \{(\texttt{M}_i, xc_i) \mid i \in I\}}{\texttt{M} \xrightarrow{\texttt{Prg}, \bot} D, \texttt{module}\ \Uparrow(\texttt{M}, \bot);\ \texttt{export}\ *;\ \overline{\texttt{import}\ *\ \texttt{from}\ \Uparrow(\texttt{M}_i, xc_i)}\ \Uparrow(\texttt{Prg}, \texttt{M}, \texttt{MdlC}, \bot)}$$

$$\frac{\mathsf{mdlNotUnique}(\texttt{Prg}, \texttt{M}) = \overline{\texttt{Defn}} \qquad \mathsf{mdlDelta}(\texttt{Prg}, \texttt{M}, \pi) = \overline{\texttt{Dlt}} \qquad (\overline{\texttt{Dlt}}, \overline{\texttt{Defn}}) \rightarrow^* \overline{\texttt{Defn'}} \qquad \sigma = \mathsf{genP}(\texttt{Prg}, \texttt{M}, \pi) \qquad \sigma(\overline{\texttt{Defn'}}) \rightsquigarrow^* \overline{\texttt{Defn''}} \qquad \Downarrow(\texttt{Prg}, \texttt{M}, \overline{\texttt{Defn''}}, \bot) = D = \{(\texttt{M}_i, xc_i) \mid i \in I\}}{\texttt{M} \xrightarrow{\texttt{Prg}, \pi} D, \texttt{module}\ \Uparrow(\texttt{M}, \pi);\ \texttt{export}\ *;\ \overline{\texttt{import}\ *\ \texttt{from}\ \Uparrow(\texttt{M}_i, xc_i)}\ \Uparrow(\texttt{Prg}, \texttt{M}, \overline{\texttt{Defn''}}, \pi)}$$

The first rule generates a module implementing the unique part of the variants of a given VM $\texttt{M}$. To do so, it extracts the unique part $\overline{\texttt{Defn}}$ of the VM , its optional init block, and the dependencies D occurring in these parts. The rule returns the set of dependencies D (which identifies variability-free ABS modules that need to be generated) and a new variability-free ABS module named $\Uparrow(\texttt{M}, \bot)$ that: (i) exports everything; (ii) imports from all variability-free ABS modules identified in D; and (iii) contains the unique classes/interfaces of the original VM, where all syntactic dependencies have been translated according to $\Uparrow$.

The second rule generates a variability-free ABS module implementing the non-unique part of the variant of the VM $\texttt{M}$ for the product π. It is similar to the first rule, except for two elements: (i) the optional init block is not considered (since it cannot be present); (ii) the extracted (non-unique classes/interfaces) part of the VM is modified by applying the activated deltas as described in Def. 9 before being integrated in the resulting module. The next definition gives the rewrite rules for flattening a whole ABS-VM program.

Definition 12 (Flattening an ABS-VM program). *Let ϵ denote the empty program, representing the initial partial result of flattening an ABS-VM program* Prg. *The rules define a judgment of the form* Prg', $A_1, D_1 \xrightarrow{\texttt{Prg}}$ Prg'', A_2, D_2, *where:* Prg' *(either ϵ or a variability-free ABS program) is a partial result of the flattening of* Prg*; the set A_1 identifies the already generated variability-free ABS modules; the set D_1 identifies the variability-free ABS modules that must be generated to fulfill the dependencies in* Prg'*; the variability-free ABS program* Prg'' *is obtained by adding to* Prg' *the code of one of the variability-free ABS modules identified by D_1; the sets A_2 and D_2 are obtained by suitably updating A_1 and D_2, respectively. Let $\xrightarrow{\texttt{Prg}}{}^*$ be the transitive closure of $\xrightarrow{\texttt{Prg}}$. The flattening of ABS-VM program* Prg *is the variability-free ABS program* Prg' *such that $\epsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg'}, A, \emptyset$ holds.*

$$\frac{\texttt{mdlInit}(\texttt{Prg}) = \texttt{M} \quad \texttt{M} \xrightarrow{\texttt{Prg},\bot} D, \texttt{Mdl} \quad A = \{(\texttt{M}, \bot)\}}{\epsilon, \emptyset, \emptyset \xrightarrow{\texttt{Prg}} \texttt{Mdl}, A, (D \setminus A)}$$

$$\frac{\texttt{Prg'} \neq \epsilon \quad (\texttt{M}, xc) \in D_1 \quad \texttt{M} \xrightarrow{\texttt{Prg},xc} D, \texttt{Mdl} \quad A_2 = A_1 \cup \{(\texttt{M}, xc)\} \quad D_2 = (D_1 \cup D) \setminus A_2}{\texttt{Prg'}, A_1, D_1 \xrightarrow{\texttt{Prg}} \texttt{Prg' Mdl}, A_2, D_2}$$

The sets A and D above refer to dependencies in the original program Prg. The first rule starts with the empty ABS program and dependency sets, adds the ABS module implementing the (unique part of the) main module of Prg and updates the dependency sets. The second rule extends Prg' by adding the ABS module required by one of the dangling dependency in D_1, and replaces the dependency sets A_1 and D_2 by their updated versions A_2 and D_2. Examples of applying the flattening procedure are given in Appendix A.2.

Definition 13 (Delta-application Soundness). *Let* M *be a VM of* Prg*. Then* M *is* delta-application sound *in* Prg*, if for all $xc \in \{\bot\} \cup \{\pi \mid \pi$ is a product of M$\}$ there exists a VM Core* MdlC *and a set D such that $\texttt{M} \xrightarrow{\texttt{Prg},xc} D, \texttt{MdlC}$ holds.* Prg *is* delta-application sound*, if all VMs* M *in* Prg *are delta-application sound.*

Recall that programs are considered equal modulo permutation of class/interface definitions, field/method definitions, etc.

Theorem 2 (Flattening Soundness). *If program* Prg *in normal form adheres to the PEV and is delta-application sound, then $\epsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg'}, A, \emptyset$ for some variability-free ABS program* Prg'*.*

Proof (Sketch). First we note that the rules in Def. 12 can be applied only a finite number of times, because the set of possible dependencies in Prg is finite (bounded by the set of variants per module). Thus termination is ensured.

The fact that Prg is in normal form guarantees: (i) all VM dependencies are defined in Prg; (ii) all configuration expressions KE in syntactic dependencies are valid products of the corresponding VM. These two facts ensure that all dependencies in Prg correspond to an actual dependency $(\texttt{M}, xc)$ where M is declared in Prg and xc is either $\bot$ or a product of M. In particular, if we consider any rewriting sequence $\epsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg'}, A, D$, all pairs $(\texttt{M}, xc)$ in A, D are such that xc is either $\bot$ or a product of M. □

6 Integration into the ABS Tool Chain and Case Studies

Integration into the ABS Tool Chain We implemented the VM concept as part of the ABS compiler tool chain (with exception of open product definitions). The implementation is available at `https://github.com/abstools/abstools/tree/local_productlines`.

To integrate VMs into the ABS compiler tool chain, only the frontend (i.e., parser and preprocessor) needed to be changed. This is, because flattening (Sect. 5) produces variability-free ABS code, keeping ABS code generation and semantic analysis (type checking) as is. The ABS parser's grammar is extended with the constructs described in Sect. 3. As expected, ABS's existing delta application mechanism (including calls to `original(...)`) could be fully reused. The implementation also includes: (i) the normal form check (Def. 1), with error reporting in case it is violated (not yet fully implemented); (ii) the PEV check (Sect. 4) with error reporting in case PEV is violated; (iii) the flattening mechanism described in Sect. 5; (iv) adjustment of the feature model (needed, because VMs use a simpler feature modeling language than ABS's μTVL [5]).

Case Studies ABS-VM was used in three case studies—the source code of the studies is available at the URL given above.

AISCO (Adaptive Information System for Charity Organizations) [35], is a modular web portal that supports the business processes (information, reporting, spending, expenditure) of charity organizations. It consists of an SPL implemented in ABS and its variability reflects the differing legal and operational requirements of the organizations. The code is in production at `https://aisco.splelive.id/`. The previous architecture required co-existence of multiple ABS variants of, e.g., financial reporting. As this is not supported by classical SPL approaches, a Java framework on top of ABS handled interoperability at runtime. For the case study, the main aspects of AISCO were re-implemented in ABS-VM in 160 lines of code, one VM with four features and five different deltas for financial reporting. All variants can interoperate within one and the same program generated from the ABS-VM code.

The second case study is a portal to compare insurance services [26]. It contains a product line model with three VMs in nearly 700 lines of code with eight features.[6] It uses VMs to model different insurance offers. Their interoperability is required so that users can compare them in the portal.

The third case study is a re-modeling of the industrial `FormbaR` case study [21] (the basis of the example in Sect. 2). VMs are useful to model infrastructure elements, such as signals coming in different variants that must coexist and interoperate within the same infrastructure model. Ca. 20% of the code is amenable for remodeling with VM, up to now we remodeled ca. 4% of the overall code. The partial refactoring showed that by introducing a VM with five features (`Main, Pre, Speed, Signal, PoV`) and seven deltas, the total number of lines for the

[6] The model contains four *further* features for the legacy SPL mechanism of ABS, kept for backwards compatibility.

five remodeled kinds of signal[7] is reduced from 196 to 169 (-13%). Excluding the lines of code needed only for variability modeling (configuration knowledge and delta headers), the remodeled part has 146 lines (-25%). The original model [21] uses traits[8] to reduce code duplication in the implementation of the different kinds of signals. The ABS-VM reformulation of the model does not need to use traits. The reformulated model is: (i) shorter (in terms of code), because in the original model there is a different class for each kind of signal; and (ii) more comprehensible, because the feature model captures constraints in the model that were implicit before (e.g., that two traits should not be used by the same class class) and declaratively connects code variability to the domain model.

7 Design Decisions

Below we briefly illustrate the rationale behind the major VM design decisions.

Principle of Encapsulated Variability. The main reasons for adopting the PEV are *simplicity* and *usability*: it allows working with a standard module concept (no need for composition or disambiguation operators) and it is straightforward to find out how any object reference in a VM is implemented.

Unique Annotation. Without the `unique` keyword, unique class/interface declarations would be automatically inferred (Definition 5, Theorem 5). This creates the danger of unintended changes to the set of unique classes/interfaces of a program. Obviously, developers would still benefit from a tool that points out all the class/interface declarations that could be annotated `unique`.

Local Feature Model. Each VM has its own feature model disjoint from those of other VMs: each feature name is local to the VM where it is declared, and there is no global name space for features. Adding a global feature model connecting the local feature models might be useful (e.g., to declare a constraint that certain variants must not co-exist in the same application).

Implicit Export/Import Flattening. Each VM `M` must declare the union of the exports/imports of all its variants. Then the flattening generates the export/import clauses of each variant by dropping export clauses for classes/interfaces not present in that variant, and by creating the import clauses for the required variants of the VMs mentioned in the import clauses of `M`. This design choice avoids the programming burden of writing delta operations on export/import clauses and reduces the cognitive burden to understand VM code. We remark that to extend VMs with delta operations on export clauses, allowing to sometimes reduce export clauses, would be straightforward. On the other hand, since implicit flattening drops all unused imports, deltas on import clauses provide no advantage.

[7] *Main signals*, *presignals*, *speed limiters*, *pre-speed limiters* and *points of visibility* [21].

[8] Traits [15] are sets of methods that can be added to a class. The ABS-VM implementation supports traits. Since traits are orthogonal to the notion of VM we have not included them in the fragment of ABS-VM formalized in this paper. We refer to Damiani et al. [7] for a presentation of the notion of traits supported by ABS.

Family-based checking. The implementation of VM as part of the ABS compiler tool chain (Sect. 6) includes normal form and PEV checks before flattening. We plan to add a family-based analysis [37] to check—before flattening—whether a program `Prg` is delta-application sound (thus ensuring, according to Theorem 2, that flattening will succeed), whether the generated variability-free ABS program will be well typed, and, more generally, whether the variants of the VMs in `Prg` as a whole (including those not generated by flattening `Prg`) would form a well-typed variability-free ABS program (see Sect. 9).

8 Related Work

Schröter et al. [32] advocate the use of suitable interfaces to support compositional analysis of MPLs consisting of *feature-oriented programming*[9] (FOP) [1, Sect. 6.1], [2] for SPLs in JAVA during different stages of the development process. Damiani et al. [13] informally outlined an extension of DOP to implement MPLs of JAVA programs by proposing linguistic constructs for defining an MPL as an SPL that imports other SPLs. In their proposal the feature model and artifact base of the importing SPL is entwined with the feature models and artifact bases of the imported SPLs. Therefore, in contrast to VMs, the proposal does not support encapsulation at SPL level. More recently, Damiani et al. [11,12] formalized an extension of DOP to implement MPLs in terms of a core calculus, where products are written in an imperative version of Featherweight JAVA [18]. The idea was to lift to the SPL level the use of dependent feature models to capture MPLs as advocated by Schröter et al. [31,33]. Like the earlier paper [13], the proposed SPL construct [12] models dependencies among different SPLs at the feature model level: to use two (or more) SPLs together, one must compose their feature models. In contrast, VMs do not require feature model composition.

The proposals mentioned above do not support variant interoperabilty. Damiani et al. [9] addressed variant interoperability in the context of ABS by considering a set of product lines, each comprising a set of modules. However, encapsulation is not realized by the mechanisms at the module level (as in VMs). Instead, unique declarations are supported (unsatisfactorily) by common modules (which is not fine-grained enough), and the concepts of modularity (through modules) and variability (through product lines) are mixed up. In contrast, the VM concept proposed in this paper unifies modules and product lines by adding variability modeling directly to modules: each module is a product line, each product line is a module. This drastically simplifies the language, yet allows more far-reaching reuse of the DOP mechanism natively supported by ABS. Furthermore, VM ease the cognitive burden of variability modeling, extending a common module framework, instead of adding another layer on top of it.

[9] FOP can be characterized as the restriction of DOP, where there is a one-to-one mapping between deltas and features (each delta is activated if and only if the corresponding feature is selected), the application order is total, and there are no class/interface/field/method removal operations [30].

Kästner et al. [22] propose a *variability-aware module system* (VAMS) for procedural languages. Like in our proposal, each VAMS module is an SPL. We outline the main differences: (i) VAMS is formalized by building on a calculus in the spirit of Cardelli's module system formalization [4], where a module consists of a set of imported typed function declarations and a list of typed function definitions, and is implemented as a module system for C code. The interface of each module describes names and types of imported and exported functions, and there is a global function namespace. Even though each module has its own feature model, there is a global feature namespace. In contrast, our proposal is based on the module system of the active object language ABS [16,19] (a fairly standard module system close to JAVA and HASKELL) and is implemented as an extension of the ABS module system. Each VM has a local namespace (which reduces overhead), also features are local to VMs. (ii) VAMS variability is achieved by using an annotative approach: code elements (import/export declarations and function declarations) are annotated with presence conditions (propositional formulas over features). VM variability is achieved explicitly by DOP for class/interface declarations and implicitly for export/import declarations. (iii) To use several modules together, VAMS requires to compose them with a module composition language providing: a module composition operator (composing two modules amounts to compose two entire SPLs, including their variability); a configuration operator (that takes a module and yields a new module, where some of features have been selected or deselected); plus further operators to avoid conflicts among modules, including feature-renaming and function-renaming, as well as function-hiding operators. VM require no explicit notion of module composition and aim at simplicity and usability by adopting the PEV. (iv) The design of VAMS did not target variant interoperability (paper [22] does not mention this issue): to ensure that two variants of the same module can co-exist, it is necessary to at least create (by exploiting the module composition language) a new version of the module by renaming all its features and all its exported functions. Instead, providing usable support to variant interoperability is a central design goal of VM. (v) Kästner et al. [22] discussed and implemented approaches for providing type guarantees on configurations of all modules and all compositions. We formalized and implemented normal form and PEV checks, while we are currently working on providing type guarantees before flattening (see discussion at the end of Sect. 7).

9 Conclusion & Future Work

This work presents variability modules, a novel approach to implement MPLs, where different, possibly interdependent, variants of the same SPL can coexist and interoperate. The PEV allows to implement variability mechanisms based on standard modules.

As discussed at the end of Sect. 7, we are developing a family-based analysis [37] that, given an ABS-VM program `Prg`, checks—*without actually generating any variant*—whether all variants of the VMs in `Prg` can be generated and, as a

whole, form a well-typed variability-free ABS program (implying that `Prg` can be flattened and is well-typed). A starting point are existing family-based analyses for DOP SPLs of imperative Featherweight Java [10] and for DOP SPLs of statecharts [24] which, in turn, are inspired by a family-based analysis for FOP SPLs of Java-like programs proposed by Delaware et al. [14], as partially implemented in the AHEAD Tool Suite [36].

The VM concept can, in principle, be added to any programming language where a suitable notion of module is present (or could be added). It is worth observing that the notion of VM is orthogonal to the actual approach used to model variability. In our VM setting on top of the ABS language we use DOP, because ABS has a native DOP implementation. However, a different approach to implement variability (e.g., the annotative approach [1]) would work as well. The case studies reported in this paper suggest that VMs are a good fit for an object-based modeling language like ABS. Further investigations are needed to asses their suitability for general purpose languages like Java.

References

1. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
2. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
3. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
4. L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM.
5. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010.
6. P. Clements and L. Northrop. *Software Product Lines: Practices & Patterns*. Addison Wesley Longman, 2001.
7. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. A unified and formal programming model for deltas and traits. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2017.
8. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. Interoperability of software product line variants. In *SPLC*, pages 264–268. ACM, 2018.
9. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. Same same but different: Interoperability of software product line variants. In *Principled Software Development*, pages 99–117. Springer, 2018.
10. 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.
11. 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.
12. F. Damiani, M. Lienhardt, and L. Paolini. A formal model for multi software product lines. *Science of Computer Programming*, 172:203 – 231, 2019.

13. 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.
14. B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: A machine-checked model of safe composition. In *ESEC/FSE*, pages 243–252. ACM, 2009.
15. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2), 2006.
16. 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.
17. G. Holl, P. Grünbacher, and R. Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology*, 54(8):828 – 852, 2012. Special Issue: Voice of the Editorial Board.
18. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
19. 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.
20. 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.
21. E. Kamburjan, R. Hähnle, and S. Schön. Formal modeling and analysis of railway operations with Active Objects. *Science of Computer Programming*, 166:167–193, Nov. 2018.
22. C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In G. T. Leavens and M. B. Dwyer, editors, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 773–792, New York, NY, USA, 2012. ACM.
23. M. Lienhardt and D. Clarke. Conflict detection in delta-oriented programming. In *ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2012.
24. M. Lienhardt, F. Damiani, L. Testa, and G. Turin. On checking delta-oriented product lines of statecharts. *Sci. Comput. Program.*, 166:3–34, 2018.
25. R. Mauliadi, M. R. A. Setyautami, I. Afriyanti, and A. Azurat. A platform for charities system generation with SPL approach. In *Proc. Intl. Conf. on Information Technology Systems and Innovation (ICITSI)*, pages 108–113, New York, NY, USA, 2017. IEEE.
26. M. Mendoza. Variability-aware modules, 2020. Master's Thesis.
27. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, Berlin, Germany, 2005.
28. S. Roman. *Lattices and Ordered Sets.* Springer New York, 2008.
29. 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.
30. 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.

31. 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.
32. 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.
33. R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated analysis of dependent feature models. In *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013*, pages 9:1–9:5, 2013.
34. M. R. A. Setyautami, D. Adianto, and A. Azurat. Modeling multi software product lines using UML. In *Proc. 22nd Intl. Systems and Software Product Line Conference, vol. 1*, pages 274–278, New York, NY, USA, 2018. ACM.
35. M. R. A. Setyautami, R. R. Rubiantoro, and A. Azurat. Model-driven engineering for delta-oriented software product lines. In *26th Asia-Pacific Software Engineering Conf., APSEC, Putrajaya, Malaysia*, pages 371–377. IEEE, 2019.
36. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM.
37. 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.
38. P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study (invited paper). In *2nd International Conference on Formal Verification of Object-Oriented Software, Torino, Italy*, volume 7421 of *LNCS*. Springer, 2012.

A For Reviewers' Convenience Only

This appendix contains additional material intended for reviewers, but not for publication in the final version.

A.1 Delta Application Rules

In this appendix we present the rules describing the application of an ordered sequence of deltas $\overline{\texttt{Dlt}}$ to to a sequence of interface/class definitions $\overline{\texttt{Defn}}$ (see Definition 9).

The rules fall into the following categories:

Rules for a Sequence of Deltas The rules in Fig. 7 describe how to apply each of the deltas in a sequence. D:Empty removes the delta if no operations are left to execute. Rules D:Inter and D:Class extracts the first interface/class operation from the delta and applies it to the list of definition. D:End concludes the application process when the sequence of the deltas to be applied is empty.

Rules for a Delta The rules in Fig. 8 how to apply the actions specified by a delta to a whole class or interface definition. Rule D:AddsI adds an interface by adding its definition to the list of definitions. Rule D:RemsI removes an interface by looking up its definition using the name from the delta modifier. The rules for classes, D:AddsC and D:RemsC are analogous. Rules D:ModI and D:ModC modify an interface, or class, by applying the rules for interface modifiers (or class modifiers).

Rules for Extends/Implements Clauses The rules in Fig. 9 modify the `extends` clauses of interfaces and `implements` clauses of classes. by removing (D:EM:Rems) or adding (D:EM:Adds) it. Rule (D:EM:Empty) is applied if all modification of the clause have been applied.

Rules for Interfaces The rules in Fig. 10 modify interfaces. Rule D:I:Empty is applicable when no further modification is requested on the given interface, so that the result is the interface itself. Rule D:I:Adds adds the specified method header to the interface (provided that no header with this name is already present in the interface). Rule D:I:Rems removes an existing method header from the interface.

Rules for Classes The rules for class modification in Fig. 11 are very similar to the ones for interfaces, with two exceptions: first, manipulation of method headers is replaced by manipulation of fields (rules D:C:AddsF and D:C:RemsF) and methods implementations (rules D:C:AddsM and D:C:RemsM). Second, methods may be modified using rule D:C:Mods. This rule replace the method implementatation, but keeps the old implementation with a fresh name. If the new implementation contains an `original` statement, then this statement is replaced by a call to the old implementation.

$$\textsc{D:Empty}\quad (\texttt{delta D;}\ \overline{\texttt{Dlt}},\ \overline{\texttt{Defn}}) \to (\ \overline{\texttt{Dlt}},\ \overline{\texttt{Defn}})$$

$$\textsc{D:Class}\quad (\texttt{delta D;}\ \texttt{CO}\ \overline{\texttt{CO}}\ \overline{\texttt{IO}}\ \overline{\texttt{Dlt}},\ \overline{\texttt{Defn}}) \to (\texttt{delta D;}\ \overline{\texttt{CO}}\ \overline{\texttt{IO}}\ \overline{\texttt{Dlt}},\ (\texttt{(D;CO)} \bullet \overline{\texttt{Defn}})$$

$$\textsc{D:Inter}\quad (\texttt{delta D;}\ \texttt{IO}\ \overline{\texttt{IO}}\ \overline{\texttt{Dlt}},\ \overline{\texttt{Defn}}) \to (\texttt{delta D;}\ \overline{\texttt{IO}}\ \overline{\texttt{Dlt}},\ \texttt{(D;IO)} \bullet \overline{\texttt{Defn}})$$

$$\textsc{D:End}\quad (\varepsilon,\ \overline{\texttt{Defn}}) \to \overline{\texttt{Defn}}$$

Fig. 7. Semantics of deltas: sequence of deltas

$$\textsc{D:AddsI}\quad \frac{\mathsf{nameOf}(\texttt{ID}) \notin \mathsf{nameOf}(\ \overline{\texttt{Defn}})}{\texttt{(D;adds ID)} \bullet \overline{\texttt{Defn}} \to \texttt{ID}\ \overline{\texttt{Defn}}} \qquad \textsc{D:RemsI}\quad \frac{\mathsf{nameOf}(\texttt{ID}) = \texttt{I}}{\texttt{(D;removes I)} \bullet (\texttt{ID}\ \overline{\texttt{Defn}}) \to \overline{\texttt{Defn}}}$$

$$\textsc{D:ModsI}\quad \texttt{(D;modifies interface I EM \{ SO \})} \bullet \texttt{(interface I extends IR \{ MH \} Defn)} \to \texttt{(interface I extends (EM} \bullet \texttt{IR) \{ SO} \bullet \texttt{MH \} Defn)}$$

$$\textsc{D:AddsC}\quad \frac{\mathsf{nameOf}(\texttt{CD}) \notin \mathsf{nameOf}(\ \overline{\texttt{Defn}})}{\texttt{(D;adds CD)} \bullet \overline{\texttt{Defn}} \to \texttt{CD}\ \overline{\texttt{Defn}}} \qquad \textsc{D:RemsC}\quad \frac{\mathsf{nameOf}(\texttt{CD}) = \texttt{C}}{\texttt{(D;removes C)} \bullet (\texttt{CD}\ \overline{\texttt{Defn}}) \to \overline{\texttt{Defn}}}$$

$$\textsc{D:ModsC}\quad \texttt{(D;modifies class C EM \{ AO \})} \bullet \texttt{(class C implements IR \{ FD MD \} Defn)} \to \texttt{(class C implements (EM} \bullet \texttt{IR) \{ (D;AO)} \bullet \texttt{(} \overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{) \} Defn)}$$

Fig. 8. Semantics of deltas: single delta

$$\textsc{D:EM:Empty}\quad \varepsilon \bullet \overline{\texttt{IR}} \to \overline{\texttt{IR}} \qquad \textsc{D:EM:Adds}\quad \texttt{(adds IR' EM)} \bullet \overline{\texttt{IR}} \to \texttt{EM} \bullet \texttt{(}\overline{\texttt{IR}}\ \overline{\texttt{IR'}}\texttt{)}$$

$$\textsc{D:EM:Rems}\quad \texttt{(removes IR EM)} \bullet \texttt{(}\overline{\texttt{IR}}\ \overline{\texttt{IR'}}\texttt{)} \to \texttt{EM} \bullet \overline{\texttt{IR'}}$$

Fig. 9. Semantics of deltas: extends/implements clauses modification

$$\textsc{D:I:Empty}\quad \varepsilon \bullet \overline{\texttt{MH}} \to \overline{\texttt{MH}} \qquad \textsc{D:I:Adds}\quad \frac{\mathsf{nameOf}(\texttt{MH}) \notin \mathsf{nameOf}(\ \overline{\texttt{MH}})}{\texttt{(adds MH SO)} \bullet \overline{\texttt{MH}} \to \overline{\texttt{SO}} \bullet \texttt{(MH MH)}} \qquad \textsc{D:I:Rems}\quad \texttt{(removes MH SO)} \bullet \texttt{(MH MH)} \to \overline{\texttt{SO}} \bullet \overline{\texttt{MH}}$$

Fig. 10. Semantics of deltas: interface modification

$$\textsc{D:C:Empty}\quad \varepsilon \bullet \texttt{(}\overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)} \to \overline{\texttt{FD}}\ \overline{\texttt{MD}} \qquad \textsc{D:C:AddsF}\quad \frac{\mathsf{nameOf}(\texttt{FD}) \notin \mathsf{nameOf}(\ \overline{\texttt{FD}})}{\texttt{(D;adds FD AO)} \bullet \texttt{(}\overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)} \to \texttt{(D;}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(FD}\ \overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)}}$$

$$\textsc{D:C:AddsM}\quad \frac{\mathsf{nameOf}(\texttt{MD}) \notin \mathsf{nameOf}(\ \overline{\texttt{MD}})}{\texttt{(D;adds MD AO)} \bullet \texttt{(}\overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)} \to \texttt{(D;}\ \overline{\texttt{AO}} \bullet \texttt{(}\overline{\texttt{FD}}\ \ \overline{\texttt{MD}}\ \overline{\texttt{MD}}\texttt{)}}$$

$$\textsc{D:C:RemsF}\quad \texttt{(removes FD AO)} \bullet \texttt{(FD}\ \overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)} \to \texttt{(D;}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(}\overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)}$$

$$\textsc{D:C:RemsM}\quad \texttt{(D;removes MH}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(}\overline{\texttt{FD}}\ \ \texttt{MH \{}\ \overline{\texttt{S}}\ \texttt{return E;\}}\ \overline{\texttt{MD}}\texttt{)} \to \texttt{(D;)}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(}\overline{\texttt{FD}}\ \overline{\texttt{MD}}\texttt{)}$$

$$\textsc{D:C:Mods}\quad \frac{\mathsf{nameOf}(\texttt{MH}) = \mathsf{nameOf}(\texttt{MH'}) = m \qquad \overline{\texttt{S''}} = \overline{\texttt{S}}\,[\texttt{D\_}^{m}/_{\texttt{original}}] \qquad \texttt{E''=E}[\texttt{D\_}^{m}/_{\texttt{original}}] \qquad \texttt{MH''=MH'}[\texttt{D\_}^{m}/_{m}]}{\begin{array}{l}\texttt{(D;modifies MH \{}\ \overline{\texttt{S}}\ \texttt{return E;\}}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(}\overline{\texttt{FD}}\ \ \texttt{MH'\{}\ \overline{\texttt{S'}}\ \texttt{return E';\}}\ \overline{\texttt{MD}}\texttt{)} \\ \quad \to \texttt{(D;}\ \overline{\texttt{AO}}\texttt{)} \bullet \texttt{(}\overline{\texttt{FD}}\ \ \texttt{MH \{}\ \overline{\texttt{S''}}\ \texttt{return E'';\}}\ \ \texttt{MH''\{}\ \overline{\texttt{S'}}\ \texttt{return E';\}}\ \overline{\texttt{MD}}\texttt{)}\end{array}}$$

Fig. 11. Semantics of deltas: class modification

A.2 Flattening Examples

In this section we demonstrate the flattening procedure presented in Sect. 5 by a couple of examples.

Example 1 (Flattening the railway station MPL). Consider the railway station MPL mockup example presented in Sect. 2. It comprises the VMs:

- `RailwayStation` in Fig. 5,
- `InterlockingSys` in Fig. 4,
- `Switches` in Fig. 3, and
- `Signals` in Fig. 2.

The variability-free ABS modules generated by flattening are given in Figs. 12, 13, 14 and 15. For each VM `M` the generated module for the unique part keeps name `M`, while the generated module for product π is given a name crated by concatenating the name `M_` with the features in π, listed in lexicographical order and separated by the symbol _. More formally, we define:

- $\Uparrow(\texttt{M}, \bot) = \texttt{M}$, and
- $\Uparrow(\texttt{M}, \pi) = \texttt{M\_}F_1\texttt{\_}\ldots\texttt{\_}F_n$, where $\pi = \{F_1,\ldots,F_n\}$ $(n \geq 0)$ and the features F_i $(1 \leq i \leq n)$ are listed in lexicographic order.

Note that there are no `InterlockingSys_` and `InterlockingSys_DirOut_Modern` modules and no `Signals_Dir_Light` module since they are not required.

```
1 module RailwayStation;
2 import * from InterlockingSys;
3 import * from InterlockingSys_DirOut;
4 import * from InterlockingSys_Modern;
5 init {
6   InterlockingSys.IILS ils1 = new InterlockingSys_DirOut.CILS();
7   InterlockingSys.IILS ils2 = new InterlockingSys_Modern.CILS();
8 }
```

Fig. 12. Flattening of the main module `RailwayStation` from Fig. 5.

```
module InterlockingSys;
export *;
import * from Signals; import * from Switches;
interface IILS {  }

module InterlockingSys_DirOut;
export *;
import * from Signals;
import * from Switches;
import * from Switches_Electric;
import * from Switches_Mechanic;
import * from Signals_Light;
import * from Signals_Form;
import * from Signals_Dir_Form;
class CILS {
   Bool testSig() {
         Switches.ISwitch swNormal = new Switches_Electric.CSwitch();
         Switches.ITrack track     = new Switches_Mechanic.CTrack();
         Switches.ISwitch swNew    = track.appendSwitch();
         Signals.ISig sigNormal    = new Signals_Light.CSig();
         Signals.ISig sigShunt     = new Signals_Form.CSig();
         return sigNormal.eqAspect(sigShunt);
   }
   Switches.ISwitch createStwith() { return new Switches_Mechanic.CSwitch(); }
   Signals.ISignal createOutSignal() {
       return new Signals_Dir_Form.CSignal();
   }
   Signals.ISignal createInSignal() { return new Signals_Form.CSignal(); }
}

module InterlockingSys_Modern;
export *;
import * from Signals;
import * from Switches;
import * from Switches_Electric;
import * from Switches_Mechanic;
import * from Signals_Light;
import * from Signals_Form;
class CILS {
   Bool testSig() {
         Switches.ISwitch swNormal = new Switches_Electric.CSwitch();
         Switches.ITrack track     = new Switches_Mechanic.CTrack();
         Switches.ISwitch swNew    = track.appendSwitch();
         Sginals.ISig sigNormal   = new Signals_Light.CSig();
         Signals.ISig sigShunt    = new Signals_Form.CSig();
         return sigNormal.eqAspect(sigShunt);
   }
   Switches.ISwitch createStwith() {return new Switches_Electric.CSwitch();}
   Signals.ISignal createOutSignal() {return new Signals_Light.CSignal();}
   Signals.ISignal createInSignal() {return new Signals_Light.CSignal();}
}
```

Fig. 13. Flattening of VM `InterlockingSys` from Fig. 4.

```
module Switches;
export *;
interface ISwitch {  }
interface ITrack { Switches.ISwitch appendSwitch(); }

module Switches_Electric;
export CTrack, CSwitch;
import * from Switches;
class CSwitch implements Switches.ISwitch {  }
class CTrack implements Switches.ITrack {
    Switches.ISwitch appendSwitch() {
       Switches.ISwitch sw = new Switches_Electric.CSwitch();
       return sw;
    }
}

module Switches_Mechanic;
export CTrack, CSwitch;
import * from Switches;
class CSwitch implements Switches.ISwitch {  }
class CTrack implements Switches.ITrack {
    Switches.ISwitch appendSwitch() {
        Switches.ISwitch sw = new Switches_Mechanic.CSwitch();
        return sw;
    }
    Bool isMechanic(){return True;}
}
```

Fig. 14. Flattened VM `Switches` from Fig. 3.

```
module Signals;
export *;
interface ISig { Bool eqAspect(ISig);  Unit setToHalt(); }

module Signals_Light;
export LSig, CSig;
import * from Signals;
class CSig implements Signals.ISig {
    Unit addBulb(){ new Signals_Light.CBulb();}
}
class CBulb {};

module Signals_Form;
export LSig, CSig;
import * from Signals;
class CSig implements Signals.ISig {}

module Signals_Dir_Form;
export LSig, CSig;
import * from Signals;
class CSig implements Signals.ISig {}
interface IDir{}
```

Fig. 15. Flattening of VM `Signals` from Fig. 2.

Example 2 (Wrong use of non-`unique`* interfaces).* Consider the code in Fig. 16. It is *not* a valid ABS-VM program and the flattened code is rejected by the ABS type checker. In the flattened code (given in Fig. 17) there are two interfaces `I`: one in module `M0_f1` for the specific part of the variant for `P1` and similar in `M0_f2` for `P2`. Even though the two interface declarations are syntactically equal, none is a subtype of the other. Therefore, the assignment `i1 = i2`, where `i1` has type `M0_f1.I` and `i2` has type `M0_f2.I`, is ill typed in generated variability-free ABS program. The family-based analysis that we are currently developing (see the discussion at the end of Sect. 7) will detect the error in the ABS-VM code, before performing flattening.

```
1  module M0;
2  export I,D; features f1,f2 with f1 <-> !f2;
3  product P1 = {f1}; product P2 = {f2};
4  class D implements I{}
5  delta Delta1; adds interface I{}
6  delta Delta2; adds interface I{}
7  Delta1 when f1; Delta2 when f2;
8
9  module Main;
10 import * from M0;
11 init {
12    I with P1 i1 = new D() with P1;
13    I with P2 i2 = new D() with P2;
14    i1 = i2; //type error
15 }
```

Fig. 16. An ABS-VM program with a wrong use of non-`unique` interfaces.

```
1  module M0_f1;
2  export I,D;
3  interface I{}
4  class D implements M0_f1.I{}
5
6  module M0_f2;
7  export I,D;
8  interface I{}
9  class D implements M0_f2.I{}
10
11 module Main;
12 import * from M0;
13 init {
14    M0_f1.I P1 i1 = new M0_f1.D();
15    M0_f2.I i2 = new M0_f2.D();
16    i1 = i2; //type error
17 }
```

Fig. 17. Flattening of the ABS-VM program in Fig. 16.