Variability Modules for Java-like Languages

Ferruccio Damiani
University of Turin
Department of Computer Science
Turin, Italy
ferruccio.damiani@unito.it

Reiner Hähnle
Technical University of Darmstadt
Department of Computer Science
Darmstadt, Germany
reiner.haehnle@tu-darmstadt.de

Eduard Kamburjan
University of Oslo
Department of Informatics
Oslo, Norway
eduard@ifi.uio.no

Michael Lienhardt
ONERA
Palaiseau, France
michael.lienhardt@onera.fr

Luca Paolini
University of Turin
Department of Computer Science
Turin, Italy
luca.paolini@unito.it

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 variability 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, provide an implementation for the Java-like modeling language ABS, and evaluate VMs by case studies.

ACM Reference Format:
Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, Michael Lienhardt, and Luca Paolini. 2021. Variability Modules for Java-like Languages. In *25th ACM International Systems and Software Product Line Conference (SPLC 2021), September 6–11, 2021, Leicester, UK*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modeling variability aspects of complex softwre systems poses challenges currently not adequately met by standard approaches to software product line engineering (SPLE) [6, 26]. 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 a multi product line (MPL) [16]. A second modeling challenge, orthogonal to MPLs, is the situation when different product variants from the same product line need to co-exist in the same context and must be interoperable [8].

In Sect. 2 we exemplify these two challenges in the context of an industrial case study from the literature [19, 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, MPLs give rise to the quest for mechanisms for hiding implementation details, reducing dependencies, controlling access to elements, etc. [16].

We take the standard concept of a module, used to structure large software systems since the 1970s, as a baseline. Software modules are supported in many programming and modeling languages, including ABS, Ada, Haskell, Java, Scala, to name just a few. Because modules are intended to facilitate 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 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. Nevertheless, there are a number of fundamental design decisions to take in the VM design. These are motivated and discussed in Sect. 3.

We formulate the VM concept as an extension of the standard concept of module for Java-like (i.e., object-oriented, class-based and strongly typed) languages. To support variability, VMs employ *delta-oriented programming* (DOP) [1, Sect. 6.6.1], [28]. Specifically, we contribute (i) a theoretical foundation of VMs, including formal syntax and semantics, in terms of a core calculus; and (ii) an implementation of VMs as an extension of the ABS language [15, 18].

We choose ABS because it features native implementations of DOP and it was successfully used in industrial case studies for variability modeling [21, 24, 37]. We stress that VMs can be added

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC 21, Leicester, UK,
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on top of *any* Java-like language. For instance, a proof-of-concept realization of VM for the Java programming language, based on architectural patterns, is (informally) described in [34]. That paper demonstrates the usefulness of the VM concept, but lacks several VM features introduced here, as well as a formal foundation.

The formal underpinnings of VMs are covered in Sects. 4–7. In Sect. 4 we declare the VM syntax and spell out consistency requirements. Sect. 5 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. 6 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 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. In Sect. 7 we prove soundness of flattening.

Sect. 8 describes how the VM concept is integrated into the existing ABS tool chain. As long as one has control over the parser and abstract syntax tree, it is relatively straightforward to realize the flattening algorithm of Sect. 6 within any compiler tool chain. We evaluate VMs by means of case studies. Related work is discussed in Sect. 9, we conclude in Sect. 10 by outlining ongoing work.

2 INTRODUCING VARIABILITY MODULES

We illustrate VMs with an example based on an industrial case study from railway engineering [21].

Our 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.[1] 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., signals have always 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 signal variants. Variability shows, for example, in the presence of an additional class `Bulb` in the light signals variant and in the fact that method `setToHalt` (which changes the shown aspect to "Halt") is different for form signals and light signals (the latter communicate with their `Bulb` instances).

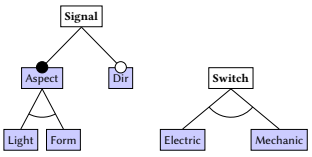

Figure 1: Features of Signals and Switches.

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

[1]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 (Fig. 1) and propositional formulas (lines with keyword **`features`** in Figs. 2–3).

```
// 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;
```

Figure 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;
```

Figure 3: An SPL of tracks and switches as a VM.

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 are declarations of interface `ISig` and of class `CSig` that implements `ISig`. By default, class and interface definitions can be modified/removed by deltas to obtain different product variants, however, 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. Class and interface declarations are followed by the deltas that describe the implementation of different variants and their application conditions. Classes and interfaces added by deltas can be modified or removed again 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

```
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 =
7     { Modern => {Electric}, !Modern => {Mechanic} }
8  product PSignal for Signals  =
9     { DirOut &&  Modern => {Light,Dir}, Modern => {Light},
10      DirOut && !Modern => {Form,Dir}, !Modern => {Form}  }
11 // CORE PART
12 unique interface IILS {  }
13 class CILS {
14   Bool testSig() {
15     ISwitch swNormal = new CSwitch() with {Electric};
16     ITrack track     = new CTrack() with {Mechanic};
17     ISwitch swNew    = track.appendSwitch();
18     ISig sigNormal   = new CSig() with LSig;
19     ISig sigShunt    = new CSig() with {Form};
20     return sigNormal.eqAspect(sigShunt);
21   }
22   ISwitch createSwitch() { return new CSwitch() with PSwitch; }
23   ISignal createOutSignal() { return new CSignal() with PSignal; }
24   ISignal createInSignal() {
25     return new CSignal() with PSignal - {Dir}; }
26 }
```

Figure 4: An SPL of interlocking systems as a VM.

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 8 contains an open product definition for the VM `Signal`. It is worth observing that open product definitions enable implementing different variants of the VM `InterlockingSys` even 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 17 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 20.

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: a VFM containing an implicit class providing an initialization method declared by 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. 4. It represents a MPL—we call it the railway station MPL.

3 DESIGN DECISIONS

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

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

Figure 5: Railway station as a main module.

Unique Annotation. As explained above, `unique` class/interface declarations in a VM `M` are shared by all variants of `M`. Without the `unique` keyword, unique class/interface declarations should be inferred (cf. Def. 7.1, Thm. 7.1 below), thus creating the danger of unintended changes of the set of unique classes/interfaces in a program. Obviously, a tool that points out all class/interface declarations that *could* be annotated `unique` would be useful.

Principle of Encapsulated Variability (PEV). The 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. 6)—which removes variability and generates those variants required by the dependencies—can resolve all dependencies among VMs to dependencies among VFM.

The main reasons for adopting the PEV are *simplicity* and *usability*: it suffices to work with a standard module concept (no need for composition or disambiguation operators as, for example, [22]) and it is easy for the modeler to find out to which implementation any object reference in a VM refers to.

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), but creates overhead.

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 need to define delta operations on export/import clauses in the language that then would be supplied by the modeler.[2] Altogether, it reduces the cognitive burden to understand VM code and the effort to write it.

Family-based checking. VM are designed to permit family-based analysis [36]. The implementation of VM as part of the ABS compiler tool chain (Sect. 8) checks PEV before flattening. Moreover, we are currently implementing a family-based analysis (described in a technical report [10]) to check—before flattening—whether a program `Prg` is delta-application sound (thus ensuring, according to Thm. 7.4, that flattening will succeed), whether the generated VFM 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 VFM.

[2] To extend VMs with delta operations on export clauses is straightforward. It would allow sometimes to shorten export clauses. On the other hand, since implicit flattening drops all unused imports, deltas on import clauses provide no advantage.

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

Figure 6: ABS-VM abridged syntax.

4 SYNTAX OF ABS VARIABILITY MODULES

The abridged syntax of *ABS with VMs* (ABS-VM, for short) is given in Fig. 6. It defines the OO fragment of ABS [15, 18], 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/product names, respectively, that are exported or imported by the VM), by the optional definition of a feature model (where `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 `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 (for example, when an interface with name `I` is imported from two different 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 (as in the final paragraph of Sect. 2) such that any other VM does not contain the keyword `init`. Observe that a VFM (as defined in the final paragraph of Sect. 2) without a product definition (a VFM may contain closed product definitions) and no occurrences of the `with` keyword is a variability-free *ABS* module.

The module delta part comprises a sequence of delta definitions `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 `CO` and by a sequence of interface operations `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 4.1 (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 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 (3a).(i) above holds; (ii) all 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'= {Φ1=> KE1,...,Φn=> KEn}` ($n \geq 1$). Then: (i) M ≠ M'; (ii) for all $1 \leq j \leq n$, all configuration/product names occurring in KE_j are imported from M'; (iii) for all $1 \leq j \leq n$, 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 = {Φ1=> KE1,...,Φn=> KEn}` ($n \geq 1$). Then: (i) for all $1 \leq j \leq n$, all configuration/product names occurring in KE_j have been already defined in M; (ii) the condition obtained from condition (3d).(iii) above by replacing M' by M holds; and (iii) condition (3d).(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), (3a), (3b), (3e) 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 (3b).(iii), (3c).(iii), (3d).(iii), (3d).(iv) and (4).(iv) can be checked with a SAT solver. Programs that cannot be transformed into normal form are rejected by the compiler.

5 ENCAPSULATED VARIABILITY

In this section we formalize the principle of encapsulated variability. We only consider programs in normal form (Def. 4.1). To formalize and to implement automated checking of PEV adherence we introduce the notion of dependency and the functions CORE, UNIQUE, and BASE.

Definition 5.1 (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 5.2 (Functions CORE, UNIQUE, BASE*).* Given a program Prg; for all VMs 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 5.3 (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 ∈ 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' ∈ UNIQUE(Prg, M).
(3) For all `with`-free-dependencies on M'.N occurring in M: if M' ≠ M then M'.N ∈ UNIQUE(Prg, M').

To check whether a program adheres to the PEV is straightforward. Programs that do not adhere to the 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 20 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 M_1 can depend on the code of a variant of a VM M_2 (and possibly vice versa). I.e., the code of M_1 refers to unique classes/interfaces of M_2 (via `with`-free-dependencies) or to classes/interfaces of a specific variant of M_2 (via `with`-dependencies). A special case of variant interdependence is when $M_1 = M_2$, i.e., M_1 has a `with`-dependency on a class/interface of M_1 itself. Then in the flattened program a variant of M_1 will contain an occurrence of a class/interface name that is declared in a different variant of M_1.

6 SEMANTICS OF VARIABILITY MODULES

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. 6.1 we introduce auxiliary functions for the extraction of relevant information from ABS-VM programs. Then, in Sect. 6.2 we give rewrite rules for transforming an ABS-VM program into a variability-free ABS program.

6.1 Auxiliary Functions

Definition 6.1 (Lookup Functions). Given a VM $\texttt{M}$ of $\texttt{Prg}$ we define the sets:

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

The meaning of `with`-free-dependencies or open `with`-dependencies δ occurring in a VM $\texttt{M}$ is relative to the product π of $\texttt{M}$ being considered. We say a dependency δ is *ground* to mean that it is either `with`-free or the configuration expression $\texttt{KE}$ in its `with`-clause is a set of features $\pi = \{\texttt{F}_1,\ldots,\texttt{F}_n\}$ $(n \geq 0)$. 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 6.2 (Ground Dependency Meaning). Given a program $\texttt{Prg}$, a VM $\texttt{M}$ of $\texttt{Prg}$, a ground dependency δ on $\texttt{M'.N}$ occurring in $\texttt{M}$, let xc be either the symbol $\bot$ or a product π of $\texttt{M}$. We define:

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

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

Flattening a program $\texttt{Prg}$ 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 $\texttt{Prg}$ 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 6.3 (New Module Name, Dependency Translation). Given a program $\texttt{Prg}$, a VM $\texttt{M}$ of $\texttt{Prg}$, let xc be either $\bot$ or a product π of $\texttt{M}$. We denote by $\Uparrow(\texttt{M}, xc)$ the name of the module that implements the unique part of the variants of $\texttt{M}$ if $xc = \bot$, or the name of the module that implements the non-unique part of the variant of $\texttt{M}$ for product π. Moreover, given a ground dependency δ on $\texttt{M'.N}$, we define:

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

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

6.2 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 $\texttt{M}$) to a sequence $\overline{\texttt{Defn}}$ of interface/class definitions (the non-unique class/interface definitions in the module core part $\texttt{Mdlc}$ of $\texttt{M}$).

Definition 6.4 (Delta Application). Given a sequence of declarations $\overline{\texttt{Defn}}$ and an ordered sequence of deltas $\overline{\texttt{Dlt}}$, we denote with the relation $(\overline{\texttt{Dlt}}, \overline{\texttt{Defn}}) \rightarrow^* \overline{\texttt{Defn'}}$ that $\overline{\texttt{Defn'}}$ is the outcome of the procedure described in App. A.

For all VMs $\texttt{M}$ of $\texttt{Prg}$, if the products π of $\texttt{M}$ are given, then the right-hand side of each open product definition `product` $\texttt{P} = \cdots$ or `product` $\texttt{P}$ `for` $\texttt{M'} = \cdots$ in $\texttt{M}$ can be evaluated to a product. This induces a mapping $\sigma = \text{genP}(\texttt{Prg}, \texttt{M}, \pi)$ from open product names to products. Given a sequence of interface/class definitions $\overline{\texttt{Defn}}$ and such a mapping σ, for at least the open product names occurring in $\overline{\texttt{Defn}}$ to products, 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 6.5 (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).

We are ready to define the rules that flatten a VM:

Definition 6.6 (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: $\texttt{Mdl}$ 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 $xc = \pi$ 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.

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

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

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 (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. 6.4 before being integrated in the resulting module. The next definition gives the rewrite rules for flattening a whole ABS-VM program.

Definition 6.7 (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 $\texttt{Prg}',A_1,D_1 \xrightarrow{\texttt{Prg}} \texttt{Prg}'',A_2,D_2$, 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_1, 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 $\varepsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg}',A,\emptyset$ holds.

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

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

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_1 by their updated versions A_2 and D_2.

7 PROPERTIES OF VARIABILITY MODULES

The following definition and theorem show that one can automatically infer the maximal set of class/interface declarations that can soundly be annotated with unique.

Definition 7.1 (Function MaxUNIQUE*).* For every program Prg, for all VM M of Prg, let $S_{\texttt{M}} = \mathsf{CORE}(\texttt{Prg},\texttt{M}) \setminus \mathsf{BASE}(\texttt{Prg},\texttt{M})$ and define $F_{\texttt{M}} : (2^{S_{\texttt{M}}},\subseteq) \to (2^{S_{\texttt{M}}},\subseteq)$ to be the non-increasing monotone function such that: $F_{\texttt{M}}(X)$ 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 $F_{\texttt{M}}(X)$ on $S_{\texttt{M}}$ until a fixpoint is reached, i.e., the set $U = F^n_{\texttt{M}}(S_{\texttt{M}})$ such that $U = F_{\texttt{M}}(U)$ for some $n \geq 0$.

Observe that MaxUNIQUE(Prg, M) is computed locally on the VM M and always terminates (since $F_{\texttt{M}}$ is non-increasing monotone and $S_{\texttt{M}}$ is finite). Unfortunately, a program Prg' such that, for all VM M of Prg', UNIQUE(Prg',M) = MaxUNIQUE(Prg',M) may not adhere to the PEV, because of item (3) in Def. 5.3. However, by the following theorem, if such a Prg' does not adhere to the PEV, then *any* program obtained from Prg' by adding or removing unique annotations does not adhere to the PEV.

Theorem 7.2 (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) ⊆ 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. 7.1 $F_{\texttt{M}}$ is a set-theoretic inclusion-preserving map and the powerset $2^{S_{\texttt{M}}}$ is a complete lattice. By the Knaster-Tarski theorem [27] there exist smallest and greatest fixpoints of $F_{\texttt{M}}$. Moreover, the PEV (Def. 5.3.(2)) requires that UNIQUE(Prg, M) is a fixpoint of $F_{\texttt{M}}$. Now item (i) holds, because MaxUNIQUE(Prg, M) is the greatest fixpoint of $F_{\texttt{M}}$—the proof is as follows: let G be the greatest fixpoint of $F_{\texttt{M}}$; clearly $G \subseteq S_{\texttt{M}}$ and (since $F_{\texttt{M}}$ is non-increasing monotone) $G = F^n_{\texttt{M}}(G) \subseteq F^n_{\texttt{M}}(S_{\texttt{M}})$ for all $n \in \mathbb{N}$; but MaxUNIQUE(Prg, M) is the fixpoint obtained by iterating $F_{\texttt{M}}$ on $S_{\texttt{M}}$. Item (ii) holds, because Prg' satisfies Def. 5.3—in particular: item (1) holds by definition of $S_{\texttt{M}}$ and non-increasing monotonicity of $F_{\texttt{M}}$; item (2) is satisfied by any fixpoint of $F_{\texttt{M}}$; since Prg satisfies item (3), so does Prg'. □

Definition 7.3 (Soundness of Delta-application). 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 7.4 (Flattening Soundness). *If program Prg in normal form adheres to the PEV and is delta-application sound, then* $\varepsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg}',A,\emptyset$ *for some variability-free ABS program Prg'.*

Proof. First we note that the rules in Def. 6.7 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 $\varepsilon,\emptyset,\emptyset \xrightarrow{\texttt{Prg}}{}^* \texttt{Prg}',A,D$, all pairs $(\texttt{M},xc)$ in A,D are such that xc is either $\bot$ or a product of M. □

8 IMPLEMENTATION AND EVALUATION

8.1 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/Edkamb/abstools/tree/variable_mod. The readme in the repository describes how to access the case studies.

To integrate VMs into the ABS compiler tool chain, only the frontend (parser and preprocessor) needed to be changed. This is, because flattening (Sect. 6.2) 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. 4. 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. 4.1), with error reporting in case it is violated (not yet fully implemented); (ii) the PEV check (Sect. 5) with error reporting in case PEV is violated; (iii) the flattening mechanism described in Sect. 6.2; (iv) adjustment of the feature model (needed, because VMs use a simpler feature modeling language than ABS's μTVL [5]).

8.2 Case Studies

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

The first case study models from scratch a portal to compare insurance services [25]. It contains a product line model with three VMs in nearly 700 lines of code with eight features.[3] It uses VMs to model different insurance offers. Their interoperability is required so that users can compare them in the portal.

The remaining three case studies either refactor or extend existing ABS models using the VM concept. Each of them illustrates a different use of VMs to support interoperable variants:

VMs vs. external tool chain. The AISCO system uses an external ad-hoc mechanism in Java to mimic variable modules in ABS. In Sec. 8.2.1 we compare the original system with our reimplementation with VMs.

VMs vs. exploiting traits. The `FormbaR` model relied on specific patterns to handle interoperability through the class model and traits. We show in Sec. 8.2.2 how the relevant parts of `FormbaR` are remodeled with VMs.

VMs vs. standard ABS SPLs. Using an ABS model of weak memory (Sec. 8.2.3), we show that without VMs, one would need to manually duplicate several modules.

8.2.1 AISCO. 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 requirements stipulate co-existence of multiple ABS variants, for example, different formats for financial reports. As this is not supported in current SPL approaches including ABS, an ad-hoc 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, instead of relying on an external, non-generic framework that is deeply interwined with the ABS model.

8.2.2 FormbaR. This is a re-modeling of the industrial `FormbaR` case study [21] (the basis of 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. The `FormbaR` model uses one class per infrastructure element, but this relies on the fact that in this case classes are a sufficient unit for variability. The domain is modelled as a tree-like type structure – additional constraints are imposed only implicitly. The part of the model that involves interoperable variability has been re-implemented using VMs.

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 five remodeled kinds of signal[4] is reduced from 241 to 180 (-25%). Excluding the lines of code needed only for variability modeling (configuration knowledge and delta headers), the remodeled part has 163 lines (-33%). The original model [21] uses traits[5] 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 length of code), because in the original model there is a separate class for each kind of signal; and (ii) more comprehensible: the feature model captures constraints in the model that were implicit before (for example, that two traits should not be used by the same class) and it declaratively connects code variability to the domain model.

8.2.3 Weak Memory Models. This case study is the VM extension of an ABS model of weak memory [20]. In sequentially consistent memory models all read- and write-accesses of some code are processed in the order they are issued. Weakly consistent (for short: weak) memory models allow partial or complete re-ordering of accesses to increase efficiency. The ABS weak memory model formalizes different relaxation strategies and device models, so as to allow to simulate and analyze their effects. An weak memory model in the case study is a class that manages a list of memory accesses. Variability includes different types of reordering (read before write, etc.). We extended the existing ABS model to two devices with two memory systems each: any of the four memory models can be different.

The ABS-VM model contains one module for memory models and one for devices (i.e., pairs of memory models). For comparison, we implemented this with product lines based on standard DOP. As we potentially need four different memory models, this required to copy the memory model module *including all deltas* four times. Furthermore, the device module had to be copied twice. Essentially, we must perform *manually* part of the VM flattening until we can rely standard operations on the modules.

The ABS-VM version has 485 LoC, of which 440 LoC are the two variable modules. The standard ABS version has 1322 LoC (+272%), of which 620 LoC are concerned with deltas and variability and 582 are the core product of the modules for memory models and devices. We refrained from reducing code duplication through traits to illustrate that product line systems without native support of interoperable variants can only replicate this behavior through massive code duplication.

If a module has p products, then in ABS-VM only p configurations are declared, for *any* number of used variants. Another observation is that to connect n variants, one needs to declare p^n products: one

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

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

[5] Traits [14] 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.

for each combination. Hence, in addition to the additional delta declarations, this blows up the feature model unreasonably.

9 RELATED WORK

On Programming Constructs for MPLs and Variant Interoperability. Schröter et al. [31] advocate the use of suitable interfaces to support compositional analysis of MPLs, consisting of *feature-oriented programming* (FOP)[6] SPLs of Java programs, 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 [17]. The idea was to lift to the SPL level the use of dependent feature models to capture MPLs, as advocated by Schröter et al. [30, 32]. 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 [9]. Setyautami et al. [33] addressed variant interoperability at the level of of static UML class diagrams. In this paper we consider executable Java-like code.

Variant interoperability in terms of ABS code is addressed in our previous work [9], where we considered a set of product lines, each *comprising* a set of modules. However, in that proposal, encapsulation is not realized by mechanisms at the module level (as in VMs): 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 interwined. In contrast, the VM concept proposed in this paper unifies modules and product lines by *adding* the capability to model variability 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, VMs ease the cognitive burden of variability modeling, extending a common module framework, instead of adding another layer on top.

On Variability-aware Module Systems. Kästner et al. [22] propose a *variability-aware module system* (that we call VAMS in the following) for procedural languages. Like in our proposal, each VAMS module is an SPL. However, there are important conceptual differences, which we outline in the following:

(i) VAMS does not encapsulate variability (cf. Sect. 5). Namely, modules import function declarations without specifying the modules from which they should be imported. In order to generate a variant, VAMS requires the user to write a composition expression, which lists all the modules to be composed and resolves dependencies and ambiguities (e.g., when a module imports a function that is defined in two different modules) by specifying how functions are renamed or hidden (and how features are renamed, selected or deselected). So, VAMS is not concerned with explicit dependencies between modules, which are crucial to usability and central to the PEV introduced in this work. By exploiting PEV, VMs achieve simplicity and usability: configuring a single VM `M` triggers automatic generation of all required variants of `M` and other VMs.

(ii) The design of VAMS does not target variant interoperability (Kästner et al. [22] do not mention this issue). Making two variants of the same module to co-exist, requires to create copy of the module and to rename (possibly by exploiting the module composition language provided by VAMS) all its features and all its exported functions. Instead, providing usable support to variant interoperability is a central design goal of VM.

(iii) VAMS variability is achieved explicitly by using an annotative approach: code elements (import/export declarations and function declarations) are annotated with presence conditions (propositional formulas over features). In VM variability is achieved explicitly by DOP for class/interface declarations and implicitly for export/import declarations.

(iv) VAMS is formalized by building on a calculus in the spirit of Cardelli's module system formalization [4] for procedural programming languages, 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. Therefore, VAMS is tailored for procedural language, where the interface of each module describes names and types of imported and exported functions, and there is a global function namespace. Moreover, even though each module has its own feature model, there is a global feature namespace. In contrast, VMs target Java-like languages, are based on the module system of ABS [15, 18] (a fairly standard module system close to Java and Haskell) and are implemented as an extension of the ABS module system. Each VM has a local namespace (which reduces overhead), also features are local to VMs.

On Variability Modules in Java. The paper [34] suggests that VM can be implemented on top of any Java-like language with modest effort. The solution[7] presented there takes a different approach from the present account: it dispenses with explicit language constructs to model variability, but uses only standard Java constructs. This is achieved with an architectural pattern: delta application is achieved with decorators, the name space is managed with abstract factories, and each product is a module declaration in itself. The sole reliance on standard Java constructs comes with some limitations: unique class/interface declarations are not directly supported (but can be achieved by suitable **final** annotations). In consequence, the PEV is not enforced. Open product declarations are not supported. Unsoundness of a product might only be detected at runtime, because reflection is used for module name resolution instead of flattening. Therefore, [34] does not feature a formal semantics of VM and family-based checking.

[6] FOP [1, Sect. 6.1], [2] 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 [29].

[7] The VM concept presented in [34] is based on the research reported here, but was published earlier due to the uncertainties inherent to peer-reviewed publication.

10 CONCLUSION AND FUTURE WORK

This work presents variability modules, a novel approach to implement MPLs consisting of DOP SPLs of Java-like programs, where different, possibly interdependent, variants of the same SPL can coexist and interoperate. The PEV allows to encapsulate variability mechanisms by standard modules.

We are currently extending the implementation (cf. Sect. 8.1) to support: (i) open product definitions; and (ii) a family-based analysis (described in a technical report [10]) 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).

In future work we would like to validate the extended tool chain by further developing the use cases (cf. Sect. 8.2) and by considering novel use cases.

A 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 6.4).

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.

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

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

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

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

Figure 7: Rules for a Sequence of Deltas

D:AddsI
$$\frac{\text{nameOf}(\texttt{ID}) \notin \text{nameOf}(\overline{\texttt{Defn}})}{(\texttt{D;adds ID}) \bullet \overline{\texttt{Defn}} \to \texttt{ID}\ \overline{\texttt{Defn}}}$$

D:RemsI
$$\frac{\text{nameOf}(\texttt{ID}) = \texttt{I}}{(\texttt{D;removes I}) \bullet (\texttt{ID}\ \overline{\texttt{Defn}}) \to \overline{\texttt{Defn}}}$$

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

D:AddsC
$$\frac{\text{nameOf}(\texttt{CD}) \notin \text{nameOf}(\overline{\texttt{Defn}})}{(\texttt{D;adds CD}) \bullet \overline{\texttt{Defn}} \to \texttt{CD}\ \overline{\texttt{Defn}}}$$

D:RemsC
$$\frac{\text{nameOf}(\texttt{CD}) = \texttt{C}}{(\texttt{D;removes C}) \bullet (\texttt{CD}\ \overline{\texttt{Defn}}) \to \overline{\texttt{Defn}}}$$

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

Figure 8: Rules for a Delta

D:EM:Empty
$$\varepsilon \bullet \overline{\texttt{IR}} \to \overline{\texttt{IR}}$$

D:EM:Adds
$$(\texttt{adds }\overline{\texttt{IR}}'\texttt{ EM}) \bullet \overline{\texttt{IR}} \to \texttt{EM} \bullet (\overline{\texttt{IR}}\ \overline{\texttt{IR}}')$$

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

Figure 9: Rules for Extends/Implements Clauses

D:I:Empty
$$\varepsilon \bullet \overline{\texttt{MH}} \to \overline{\texttt{MH}}$$

D:I:Adds
$$\frac{\text{nameOf}(\texttt{MH}) \notin \text{nameOf}(\overline{\texttt{MH}})}{(\texttt{adds MH }\overline{\texttt{SO}}) \bullet \overline{\texttt{MH}} \to \overline{\texttt{SO}} \bullet (\texttt{MH}\ \overline{\texttt{MH}})}$$

D:I:Rems
$$(\texttt{removes MH }\overline{\texttt{SO}}) \bullet (\texttt{MH}\ \overline{\texttt{MH}}) \to \overline{\texttt{SO}} \bullet \overline{\texttt{MH}}$$

Figure 10: Rules for Interfaces

D:C:Empty
$$\varepsilon \bullet (\overline{\texttt{FD}}\ \overline{\texttt{MD}}) \to \overline{\texttt{FD}}\ \overline{\texttt{MD}}$$

D:C:AddsF
$$\frac{\text{nameOf}(\texttt{FD}) \notin \text{nameOf}(\overline{\texttt{FD}})}{(\texttt{D;adds FD }\overline{\texttt{AO}}) \bullet (\overline{\texttt{FD}}\ \overline{\texttt{MD}}) \to (\texttt{D;}\overline{\texttt{AO}}) \bullet (\texttt{FD}\ \overline{\texttt{FD}}\ \overline{\texttt{MD}})}$$

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

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

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

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

Figure 11: Rules for Classes

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, R. Hähnle, E. Kamburjan, M. Lienhardt, and L. Paolini. Variability Modules and Family-based type checking for ABS. Technical report, Technical University of Darmstadt, Department of Computer Science, 04 2021. https://formbar.raillab.de/en/vm_techrep/.

[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] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2), 2006.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] E. Kamburjan and R. Hähnle. Prototyping formal system models with active objects. In M. Bartoletti and S. Knight, editors, *Proceedings 11th Interaction and Concurrency Experience, ICE 2018, Madrid, Spain, June 20-21, 2018*, volume 279 of *EPTCS*, pages 52–67, 2018.

[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] 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.

[25] M. Mendoza. Variability-aware modules, 2020. Master's Thesis.

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

[27] S. Roman. *Lattices and Ordered Sets.* Springer New York, 2008.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] M. R. A. Setyautami and R. Hähnle. An architectural pattern to realize multi software product lines in Java. In P. Grünbacher, C. Seidl, D. Dhungana, and H. Lovasz-Bukvova, editors, *Proc. 15th Intl. Working Conf. on Variability Modelling of Software-Intensive Systems, Krems, Austria*, pages 9:1–9:9. ACM Press, Feb. 2021.

[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] 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.

[37] 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.