The Journal of Systems & Software 195 (2023) 111510

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Variability modules☆

Ferruccio Damiani [a,*], Reiner Hähnle [b], Eduard Kamburjan [c,*], Michael Lienhardt [d], Luca Paolini [a]

[a] Dipartimento di Informatica, University of Turin, Turin, Italy
[b] Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany
[c] Department of Informatics, University of Oslo, Oslo, Norway
[d] ONERA, Palaiseau, France

ARTICLE INFO

Article history:
Received 21 February 2022
Received in revised form 18 July 2022
Accepted 8 September 2022
Available online 24 September 2022

Keywords:
Delta-oriented programming
Family-based analysis
Language design
Modules
Multi product line
Variant generation

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: each variant can depend on variants from other SPLs. MPLs are challenging to model and to implement efficiently, especially when different variants of the same SPL must coexist and interoperate. We address this challenge by introducing the concept of a variability module (VM), a new language construct. A VM constitutes at the same time 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 satisfy its dependencies. Consequentially, a set of interdependent VMs represents an MPL that can be compiled into a set of standard modules. We illustrate the VM concept with an example from an industrial modeling scenario and formalize it in a core calculus. We define family-based analyses to check that a VM satisfies certain well-formedness conditions and whether all variants can be generated. Finally, we provide an implementation of VM for the Java-like modeling language ABS, and evaluate it with case studies.

1. Introduction

Modeling variability aspects of complex software systems poses challenges currently not adequately met by standard approaches to software product line engineering (SPLE) (Clements and Northrop, 2001; Pohl et al., 2005). 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) (Holl et al., 2012; Trujillo-Tzanahua et al., 2018). 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 (Damiani et al., 2018a).

In Section 2 we exemplify these two challenges in the context of an industrial case study from the literature (Kamburjan and Hähnle, 2016; Kamburjan et al., 2018), performed for Deutsche Bahn Netz AG, where: (i) several interdependent product lines for networks, signals, switches, etc., occur; and (ii) 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. (Holl et al., 2012).

We take the standard concept of a module (Wirth, 1980), 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. To substantiate this claim, in Section 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 concept. In Section 3 we

☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail addresses: ferruccio.damiani@unito.it (F. Damiani), reiner.haehnle@tu-darmstadt.de (R. Hähnle), eduard@ifi.uio.no (E. Kamburjan), michael.lienhardt@onera.fr (M. Lienhardt), luca.paolini@unito.it (L. Paolini).

https://doi.org/10.1016/j.jss.2022.111510

provide background on code reuse in SPL implementation, then, in Section 4, we motivate and discuss the VM design decisions.

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) – see Schaefer et al. (2010) and (Apel et al., 2013, Sect. 6.6.1). 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 (Hähnle, 2013; Johnsen et al., 2010).

We choose ABS because it features native implementations of DOP and it was successfully used in industrial case studies for variability modeling (Kamburjan et al., 2018; Mauliadi et al., 2017; Wong et al., 2012). We stress that VMs can be added 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 by Setyautami and Hähnle (2021). 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 Sections 5–9. In Section 5 we declare their syntax and spell out consistency requirements of ABS-VM, a core calculus for ABS with VMs. Section 6 formalizes a statically checkable property of VMs: the *principle of encapsulated variability (PEV)*, which ensures that any dependency among VMs can be reduced to dependencies among standard, variability-free modules. In Section 7 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 Section 8 we define type-safety for ABS-VM programs: the guarantee that all variants of all VMs of a program can be generated and together form a well-typed variability-free ABS program. In Section 9 we define family-based analyses that partially check whether an ABS-VM program is type-safe, including the property that all variants of all VMs can be generated.

Section 10 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 family-based checks (of Section 9) and the flattening algorithm (of Section 7) within any compiler tool chain. Section 11 evaluates VMs by means of case studies. Related work is discussed in Section 12. We conclude in Section 13 by outlining ongoing work.

The present article is based on an SPLC 2021 paper (Damiani et al., 2021), with the following extensions:

1. We *define* type-uniformity, pre-typing, and applicability consistency analyses for ABS-VM programs.
2. We *implement* sanity condition checks, a PEV compliance check, maximal unique annotation inference, as well as the analyses listed in item 1
3. The VM language features so-called *open* product definitions (Section 5.1), however, these were neither *implemented* nor *used* in a case study, which is now done.
4. We *extend* the evaluation.

2. Introducing variability modules

We illustrate variability modules and how they support variant interoperability by way of an example based on the industrial

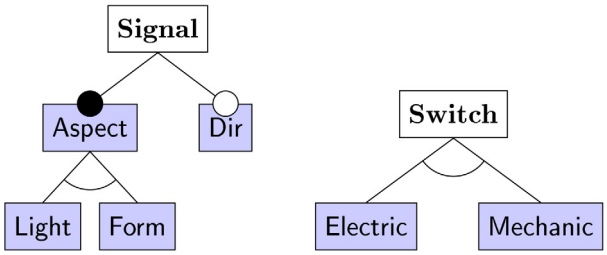

Fig. 1. Features of signals and switches.

FormbaR case study from railway engineering (Kamburjan et al., 2018).[1] We look at the full FormbaR case study during VM evaluation in Section 11.

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. Feature models (Czarnecki and Eisenecker, 2000) 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, e.g., by Batory (2005) nad Apel et al. (2013), like feature diagrams (Fig. 1) and propositional formulas (lines beginning with the keywords `features` in Figs. 2–3).

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 a signal share the same interface to the interlocking system and basic functionality, such as aspect change (for example, signals have always the aspects "Halt" and "Go"). In case of multiple outgoing tracks, a signal may also indicate the direction a train is taking — so there are four signal variants. Variability occurs, for example, in a class `Bulb`, only present in the light signals variant and in the fact that method `setToHalt` (which changes the shown aspect to "Halt") is implemented differently for form and light signals (the latter communicate with their `Bulb` instances).

Signals are modeled by the VM `Signals` in Fig. 2. It starts with the *module header*, comprising: the keyword `module`, the module *name*, the list of *exported* module elements, the *feature* list constrained with a propositional formula describing the products. This is followed by a list of *configuration* definitions (here just one), where each definition gives a name to a set of features. Finally, a list of *product* definitions (here just one), where each definition gives a *name* to a product of the SPL.

Next is the module *core part*, comprising declarations of interface `ISig` and of class `CSig` that implements `ISig`. By default, classes and interfaces can be modified/removed by deltas to obtain different product variants, however, class/interface declarations annotated by the keyword `unique` must be *the same* in all product variants. Unique declarations enable interoperability between different product variants of the same VM.

Finally there is the *delta part*, comprising 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.

[1] For presentation purposes some aspects are modeled in a slightly different manner as in Kamburjan et al. (2018).

```
// MODULE HEADER
module Signals;
export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
configuration KSig = {Dir}; //not used explicitly in this example
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 {
                adds Unit addBulb() { new CBulb();} };
delta FDelta; modifies class CSig {
                adds Date nextMotorMaintain; };
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 {
                adds Date nextMotorMaintain;};
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.

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 7 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 9 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 19 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

```
1  // MODULE HEADER
2  module InterlockingSys;
3  export *;
4  import * from Signals;
5  import * from Switches;
6  features Modern, DirOut with True;
7  product PSwitch for Switches =
8     { Modern => {Electric}, !Modern => {Mechanic} }
9  product PSignal for Signals  =
10    { DirOut &&  Modern => {Light,Dir}, Modern => {Light},
11      DirOut && !Modern => {Form,Dir}, !Modern => {Form}  }
12
13 // CORE PART
14 unique interface IILS { }
15 class CILS implements IILS {
16   Bool testSig() {
17     ISwitch swNormal = new CSwitch() with {Electric};
18     ITrack track     = new CTrack() with {Mechanic};
19     ISwitch swNew    = track.appendSwitch();
20     ISig sigNormal   = new CSig() with LSig;
21     ISig sigShunt    = new CSig() with {Form};
22     return sigNormal.eqAspect(sigShunt);
23   }
24   ISwitch createSwitch() { return new CSwitch() with PSwitch; }
25   ISignal createOutSignal() { return new CSignal() with PSignal; }
26   ISignal createInSignal() {
27     return new CSignal() with PSignal - {Dir}; }
28 }
```

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

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

Fig. 5. A railway station as a main module.

is only used inside the VM `Signals`. Different product variants are fully interoperable, as witnessed by the expression in line 22.

We conclude this brief overview of VMs by defining the terminology of unique and main module.

- A *unique module* (UM) is a VM that does not contain a feature model. Therefore, it contains no configuration definitions, no open product definitions, and no deltas, however, it may contain `with` clauses and closed product definitions. All classes and interfaces of a UM are implicitly considered unique, so the `unique` keyword can be omitted.
- A *main module* is a UM containing an implicit class providing an initialization method declared by the keyword `init`. Each program has at most one main module. Programs without a main module cannot be executed (like Java programs without a class containing a main method).

The whole railway station is modeled by the main module `RailwayStation` (Fig. 5) as well as VMs `Signals` (Fig. 2), `Switches` (Fig. 3), and `InterlockingSys` (Fig. 4). They represent an MPL called the *railway station* MPL henceforth.

3. Code reuse in SPL implementation

The example in Section 2 illustrates how DOP supports SPL implementation, and how VMs support both interoperable variants and MPLs. Before we continue with the formalization of VMs, let us briefly recall the bigger picture in software product line engineering as far as code reuse in feature-oriented SPL implementation is concerned.

In software product line engineering one aims to develop a family of similar programs, called variants, by managed reuse (Clements and Northrop, 2001; Pohl et al., 2005). In feature-oriented SPLs (Apel et al., 2013) this amounts to: (i) From the *problem space* perspective, each variant of an SPL is identified by a set of features, called a *product*; (ii) from the *solution space* perspective, code reuse mechanisms for variant implementations have to be flexible enough to capture commonalities in the variants and to realize the desired variants (Schaefer et al., 2012).

Consider, for instance, the SPL for signals in Section 2: (i) Concerning the problem space, it has n features and m products; (ii) concerning the solution space, it has programs written in a Java-like language as variants, where each variant represents a signal by means of an interface `ISig`, which is the same for all variants, and a class, which is different in each variant. A straightforward, but naive, approach to SPL implementation provides the code of all m variants within the same program. Then the SPL for signals is implemented as a program comprising the interface `ISig` and m classes `CSig1`,..., `CSig`m (one for each product). This approach, however, has several issues:

1. The code reuse mechanism of a Java-like implementation language might not be flexible enough to capture commonalities in the variants, so some code might be duplicated. In particular, the rigidity of class-based inheritance (code reuse can be realized only within a class hierarchy Mikhajlov and Sekerinski, 1998; Ducasse et al., 2006) puts limitations on the reuse of code across variants.
2. Java-like languages possess no construct to explicitly capture the relation between the problem and the solution spaces. To document the relation between features and variants, and to support automatic variant generation, *ad hoc* external tool support must be added.
3. The approach does not scale: recall that an SPL with n features can have up to $m = 2^n$ products, thus up to 2^n classes `CSig`i must be defined.

Using a Java-like language featuring flexible code reuse mechanisms (for example, traits (Schärli et al., 2003; Ducasse et al., 2006)[2] or the constructs of the JIGSAW framework (Bracha,

[2] Traits were first proposed in the dynamically typed, object-oriented, class-based Smalltalk/Squeak language (Schärli et al., 2003; Ducasse et al., 2006) and

1992)[3]) addresses the first issue, but not the remaining ones. To address the second and third issue, programming constructs specifically designed for *inter-product* (i.e., across different variants) code reuse have been developed.

Following Schaefer et al. (2012), we classify programming constructs dedicated to inter-product code reuse into three approaches. First, the *annotative approach* expresses negative variability: code is marked with those features of an SPL it implements and a variant is obtained by excluding code associated with *non-selected* features. An example of the annotative approach is CIDE (Kästner et al., 2012a). Second, the *compositional approach* expresses positive variability: variants are built by composing code fragments that implement specific features. A prominent instance is *feature-oriented programming* (FOP), for example Batory et al. (2004) or (Apel et al., 2013, Sect. 6.1). FOP can be viewed as a restriction of DOP with the following constraints: 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 delta application order is total, and there are no class/interface/-field/method removal operations (Schaefer and Damiani, 2010). Third, the *transformational approach* expresses both positive and negative variability. DOP is an instance of the transformational approach.

In the context of annotative/compositional/transformational approaches to inter-product code reuse, the code reuse mechanism provided by the language in which each variant is written has the role of an *intra-product* code reuse mechanism (that is, it expresses code reuse *within* the same variant) (Damiani et al., 2014a).

Remark 1 (*Inter- and Intra-Product Code Reuse in ABS*)**.** The original version of the ABS language (Hähnle, 2013; Johnsen et al., 2010) has deltas as a construct for inter-product code reuse and *no construct* for intra-product code reuse: we refer to this version of ABS as *Delta-ABS*. Hence, in some Delta-ABS case studies code duplication within the same variant was present. A notable case of a Delta-ABS model with duplicated code is the first version of the `FormbaR` case study (Kamburjan and Hähnle, 2016), where each infrastructure element (for example, a signal) is modeled by a class and (since different variants of the same infrastructure element need to coexist) a separate class is declared for each variant of an infrastructure element. Certain code is duplicated across all variants, for example, the code for a method that models the end of a train passing through an infrastructure element is duplicated in three out of five classes present.

A more recent ABS version (Damiani et al., 2017a) added traits for intra-product code reuse to Delta-ABS: we refer to that ABS version with *Delta-Trait-ABS*. The second version of the `FormbaR` case study (Kamburjan et al., 2018), implemented in Delta-Trait-ABS, saves duplicated code by using traits, resulting in 40% fewer lines of code on the named infrastructure alone. The second and third issue noted above, however, are not addressed with traits.

To the best of our knowledge, no programming construct for implementing interoperable variants overcomes the second and third issue discussed above. VMs address all three issues and provide proper support for MPLs as well as variant interoperability. In Section 11 we present a refactored version of the `FormbaR` case study based on VMs.

subsequently formulated in a Java-like setting by various authors (Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Reppy and Turon, 2007; Bono et al., 2008; Liquori and Spiwack, 2008; Bettini et al., 2013b; Bettini and Damiani, 2017).

[3] The constructs of the JIGSAW framework (Bracha, 1992) were formalized in a Java-like setting by means of the FGig calculus (Lagorio et al., 2009, 2012).

4. Design decisions

We briefly illustrate the rationale behind the major VM design decisions.

Unique Annotation. As illustrated in Section 2, `unique` class/interface declarations in a VM `M` are shared by all variants of `M`. Without the `unique` keyword, unique class/interface declarations need to be inferred (via Definition 4), creating the danger of unintended changes of the set of classes/interfaces in a program considered to be unique. Obviously, a tool that points out all class/interface declarations that *could* be annotated `unique` is 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 Section 7) – which removes variability and generates those variants required by the dependencies – can resolve all dependencies among VMs to dependencies among UMs. 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, Kästner et al. (2012b)) and it is easy for the modeler to find out to which implementation any object reference in a VM refers to.

Disjoint Feature Models. Each VM has its own feature model, disjoint from those of other VMs: each feature name belongs to the VM where it is declared (like class interface names). There are no export/import clauses for features: each VM can refer to a feature `f` of another VM `M` only within a scope where the name `M` is specified: each occurrence of a feature name `f` is understood as `M.f` for a VM name `M` that is clear from the context. Feature names can be defined, with a different meaning, in different VMs. Feature names are implicitly exported/imported. It might be useful to add support for expressing constraints connecting the different feature models (e.g., to specify that certain variants must not co-exist in the same application).

Implicit Export/Import Flattening. Each VM `M` must declare any export/ import that may occur in one of its variants. The flattening generates more specific export/import clauses for each variant by dropping the export clauses for classes/interfaces not present in that variant, and by creating 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.[4] Altogether, it reduces the cognitive burden to understand VM code and the effort to write it.

Family-based Checking. VMs are designed to facilitate family-based analyses (Thüm et al., 2014). The implementation of VMs as part of the ABS compiler tool chain (Section 10) supports family-based analysis to check – before flattening

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

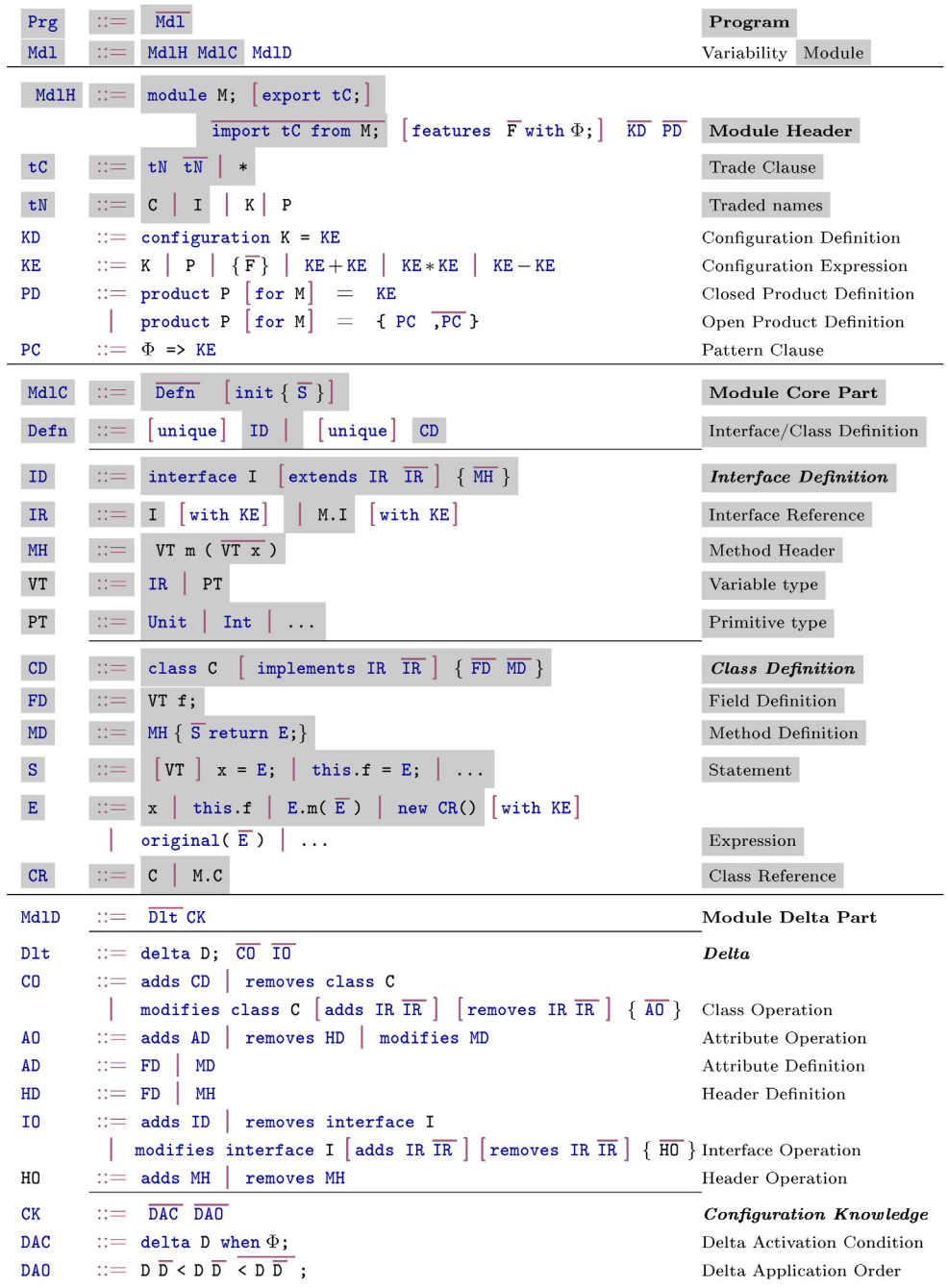

Fig. 6. ABS-VM abridged syntax — the fragment of the grammar describing the syntax of Vf-ABS is highlighted in gray.

– whether a program `Prg` satisfies the PEV and whether all variants of all VMs in `Prg` can be generated and would form (as a whole) a variability free ABS program that satisfies certain well-formedness conditions (Section 9).

5. Syntax of variability modules

Fig. 6 defines the syntax of *ABS with VMs* (ABS-VM, for short) as an extension of *variability-free ABS* (Vf-ABS, for short). Vf-ABS is the sequential, OO fragment of ABS (Johnsen et al., 2010; Hähnle, 2013) and ABS-VM extends it with VM concepts.[5]

We write `M` to denote module names, `C` for class names, `I` for interface names, `K` for feature configuration names, `P` for product names, `D` for delta names, `m` for method names, and `f` for field names. A *named element* is either a (variability) module, an interface/class definition, a method header, a field/method definition, a method formal parameter declaration, a delta, a class/attribute/interface/header operation, or a delta activation condition. Given a named element X, we write $\mathsf{name}(X)$ to denote its name. Following Igarashi et al. (2001), $\overline{X}$ denotes a possibly empty finite sequence of elements X. If the elements in $\overline{X}$ are named, we assume their names are pairwise distinct.

[5] Even though ABS supports variability management based on deltas, for presentational purposes, we singled out Vf-ABS. We stress that the syntax of delta definitions in ABS-VM is essentially identical to ABS syntax.

5.1. ABS-VMsyntax

A program is a sequence of VMs. 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 $\overline{\texttt{F}}$ 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 feature sets ($+$, $*$ 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 applicable clause is found.[6] 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 (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.

Remark 2 (*On UMs and Vf-ABS Modules*)**.** A UM, as defined in Section 2, is a Vf-ABS module if and only if it does not contain product definitions or occurrences of the `with` keyword.

The 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 its previous incarnation via the reserved method name `original`.

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 Bettini et al., 2013a). 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 (Bettini et al., 2013a; Lienhardt and Clarke, 2012). Without loss of generality, for each VM, we assume that the total order in which delta definitions are listed is compatible with R.

[6] The sanity conditions (see Section 5.2) guarantee that such a clause exists.

5.2. ABS-VM sanity conditions

In the rest of the paper, without loss of generality, we assume each ABS-VM program Prg to satisfy a number of *sanity conditions* that enforce minimal consistency requirements for ABS-VM programs. These sanity conditions, listed below, apply to each VM name M occurring anywhere in Prg.

1. Prg contains a definition of the VM M.
2. If M is a main module (see end of Section 2) then any other VM of Prg does not contain the keyword `init` (i.e., Prg contains at most one main module).
3. If M contains a clause `import tC from M'` then $\texttt{M'} \neq \texttt{M}$ and all traded names in tC occur in the export clause of M'.
4. All class references CR and interface references IR occurring in M are qualified, that is of the form `M'.N [with ...]` for some module name M' and class/interface name N such that if $\texttt{M'} \neq \texttt{M}$ then M contains the import clause `import tC from M'` where either `tC` = $*$ or N occurs in tC.
5. All traded names occurring in the export clause of M are defined in M: each configuration/product name is defined in the module header and each class/interface name is defined in the module core part or added by some delta in the module delta part.
6. If M is a UM (see end of Section 2) then all its class/interface definitions have the qualifier `unique`.
7. No open product names are exported and no product names with a declaration of the form `product P for M'=` $\cdots$ are exported.
8. The following properties hold for different definitions in M:
   (a) `configuration K = KE`: (i) all feature names occurring in KE are features of M and all configuration names occurring in KE have been already defined in M; (ii) no product name occurs in KE.
   (b) `product P = KE`: (i) condition (8a). (i) above holds; (ii) all product names occurring in KE have been already defined in M and are closed; (iii) KE denotes a product of M.
   (c) `product P for M'= KE`: (i) $\texttt{M} \neq \texttt{M'}$; (ii) all feature names occurring in KE are features of M' and all configuration/product names occurring in KE are imported from M'; (iii) KE denotes a product of M'.
   (d) `product P for M'= {`Φ_1`=> KE`$_1$`, ..., `Φ_n`=> KE`$_n$`}` ($n \geq 1$): (i) $\texttt{M} \neq \texttt{M'}$; (ii) for all $1 \leq j \leq n$, all configuration/product names occurring in $\texttt{KE}_j$ are imported from M'; (iii) for all $1 \leq j \leq n$, all feature names occurring in Φ_j are features of M and all feature names occurring in $\texttt{KE}_j$ are features of M'; (iv) for all $1 \leq j \leq n$, at least one product of M satisfies $(\bigwedge_{1 \leq i < j} \neg \Phi_i) \wedge \Phi_j$, and $\texttt{KE}_j$ denotes a product of M'; (v) all products of M satisfy $\bigvee_{1 \leq i \leq n} \Phi_i$.
   (e) `product P = {`Φ_1`=> KE`$_1$`, ..., `Φ_n`=> KE`$_n$`}` ($n \geq 1$): (i) for all $1 \leq j \leq n$, all configuration/product names occurring in $\texttt{KE}_j$ have been already defined in M; (ii) the condition obtained from condition (8d). (iii) above by replacing M' with M holds; (iii) the condition obtained from condition (8d). (iv) above by replacing M' with M holds; (iv) condition (8d). (v) above holds.

9. If M contains an occurrence of `new M'.C() with KE` or `M'.I with KE`: (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' = ...`; (iv) either KE does not contain occurrences of open product names or it is an open product name; (v) KE denotes a product of M'.

To check whether a program satisfies the sanity conditions is straightforward: for each VM M:

- Conditions (1), (2) can be checked by inspection of Prg.
- Condition (3) can be checked by inspection of M and of the header of the modules M' listed in the import clauses of M.
- Conditions (4)–(7), (8a), (8b), (8c)(i), (8d)(i), (8e)(i)–(ii), (9)(i)–(ii) can be checked by inspection of M, where condition (8b)(iii) also requires to evaluate a propositional formula.
- Conditions (8c)(ii)–(iii), (8d)(ii)–(iii), and (9)(iii)–(iv) can be checked by inspection of M and of the header of M', where conditions (8c)(iii) and (9)(v) also requires to evaluate a propositional formula.
- Conditions (8d)(iv)–(v), (8e)(iii)–(iv) can be checked with a SAT solver (Alouneh et al., 2019). We call them *SAT sanity conditions*.

It is worth observing that sanity conditions (1)–(5) apply as well to Vf-ABS programs. When they are satisfied for a given Vf-ABS program we say it is *sane*.

6. Encapsulated variability

In Section 6.1 we formalize the *principle of encapsulated variability (PEV)* that was informally introduced in Section 4. In Section 6.2, we address the issue of inferring the maximal set of class/interface declarations that can soundly be annotated with `unique`.

6.1. PEV-compliance

To formalize PEV-compliance, we first introduce some auxiliary definitions. Namely, the notion of dependency, and the functions CORE, UNIQUE and BASE.

Definition 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).

- A dependency of the form `... with KE` is called `with`-*open-dependency* if KE contains an occurrence of an open product name P, it is called `with`-*closed-dependency* otherwise.
- A dependency is called *ground* if it is: either `with`-free or of the form `... with` π, where π is a set of features $\{F_1,\ldots,F_n\}$ ($n \geq 0$).

Given the notion of dependency, we now define the CORE function, which returns the set of class and interfaces defined in the core part of a VM, the UNIQUE function returning the subset of the core that is annotated as `unique`, and the BASE function returning the subset of the core that is modified by some delta.

Definition 2 (*Functions* CORE, UNIQUE, BASE)**.** Given a program Prg, then 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) $\subseteq$ CORE(Prg,M) contains those class/interface names whose declaration is annotated with `unique`; BASE(Prg,M) $\subseteq$ CORE(Prg,M) contains those class/interface names that are modified, removed or added by some delta of M.

We can now formalize PEV-compliance in terms of the above functions as follows.

Definition 3 (*PEV-Compliance*)**.** A program Prg is PEV-compliant iff for all VMs M of Prg:

1. UNIQUE(Prg,M) $\cap$ BASE(Prg,M) $= \emptyset$.
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 is PEV-compliant is straightforward and programs that are not PEV-compliant are rejected by the compiler. The code in Section 2 is PEV-compliant.

Example 1 (*Lack of PEV-Compliance*)**.** Consider the three programs in Fig. 7. None of them is PEV-compliant. The first program Prg_1 violates the first condition: I is part of UNIQUE(Prg_1,M) (because of its annotation) and of BASE(Prg_1,M) (because it is modified by D).

The second program Prg_2 violates the second condition: M.I has a dependency on M.J that is not unique: M.J $\notin$ UNIQUE(Prg_2,M). Because of this, it is not determined which interface is extended, as there may be multiple variants even for dependencies *within the same module*. The third program Prg_3 is analogous for dependencies to other modules and violates the third condition, as M′.I $\notin$ UNIQUE(Prg_3,M′)

According to the PEV, VMs support two types of interaction among variants:

Variant interoperability. Different variants of the *same* VM can co-exist and cooperate via unique classes/interfaces. For instance, in the railway station MPL of Section 2, all *interfaces* are unique and all *classes* are not unique (which is a common pattern). Then, in line 22 of Fig. 4, an instance of class CSig in the variant of VM Signal for product {Light} receives an invocation of method eqAspect that declares an argument of type Signal 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.

6.2. Maximal set of `unique` annotations

The following definition and theorem show that, given an ABS-VM program that satisfies the sanity conditions (see Section 5.2) one can automatically infer the maximal set of class/interface declarations that can soundly be annotated with `unique`.

$\mathtt{Prg}_1$:
```
module M;
unique interface I {}
delta D; modifies interface I { adds Unit m(); }
```

$\mathtt{Prg}_2$:
```
module M;
unique interface I extends J {}
interface J {}
delta D; modifies interface J { adds Unit m(); }
```

$\mathtt{Prg}_3$:
```
module M;
import * from M';
interface J extends I {}

module M'; export *;
interface I {}
delta D; modifies interface I { adds Unit m(); }
```

Fig. 7. Not PEV-compliant programs.

Definition 4 (*Function* MaxUNIQUE)**.** For every ABS-VM program Prg, for all VM M of Prg, let $S_{\mathtt{M}}$ = CORE(Prg, M) \ BASE(Prg, M) and $F_{\mathtt{M}}$: $(2^{S_{\mathtt{M}}}, \subseteq) \rightarrow (2^{S_{\mathtt{M}}}, \subseteq)$ is defined as the non-increasing monotone function such that: $F_{\mathtt{M}}(X)$ is the subset of X obtained by removing simultaneously 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. $U = F^n_{\mathtt{M}}(S_{\mathtt{M}})$ such that $U = F_{\mathtt{M}}(U)$ for some $n \geq 0$.

Function MaxUNIQUE(Prg, M) is computed locally on the VM M and always terminates (since $F_{\mathtt{M}}$ is non-increasing monotone and $S_{\mathtt{M}}$ 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 Definition 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 1 (*Maximal Set of Unique Annotations Admitting 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 Definition 4 $F_{\mathtt{M}}$ is a set-theoretic inclusion-preserving map and the powerset $2^{S_{\mathtt{M}}}$ is a complete lattice. By the Knaster–Tarski theorem (Roman, 2008) there exist smallest and greatest fixpoints of $F_{\mathtt{M}}$. Moreover, the PEV (Definition 3.(2)) requires UNIQUE(Prg, M) to be a fixpoint of $F_{\mathtt{M}}$. Now item (i) holds, because the set 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 $S_{\mathtt{M}}$. Item (ii) holds, because Prg' satisfies Definition 3—in particular: item (1) holds by definition of $S_{\mathtt{M}}$ and non-increasing monotonicity of $F_{\mathtt{M}}$; item (2) is satisfied by any fixpoint of $F_{\mathtt{M}}$; since Prg satisfies item (3), so does Prg'. □

7. Flattening semantics of variability modules

In this section, we consider (without loss of generality) sane ABS-VM programs that are in normal form, according to the following definition.

Definition 5 (*ABS-VM Normal Form*)**.** An ABS-VM program Prg is in *normal form* if: (i) Prg is PEV-compliant; (ii) all configuration definitions and closed product definitions are *resolved* in Prg, i.e.:

- All corresponding declarations are removed from Prg.
- All occurrences of names of such definitions are removed from export/import clauses of Prg.[7]
- All occurrences of such definitions in Prg are replaced with their value (a set of features)[8] and all the occurrences of with KE such that KE is not an open product name are replaced with with π, where π is the value of KE (a product).[9]

In Section 7.1 we introduce auxiliary functions for the extraction of relevant information from ABS-VM programs. In Section 7.2 we give the semantics of ABS-VM in terms of rewrite rules for transforming an ABS-VM program into a Vf-ABS program.

7.1. Auxiliary functions

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

- mdlUnique(Prg, M) for all interface/class definitions in the Module Core Part of M annotated with unique.
- mdlNotUnique(Prg, M) for all interface/class definitions not contained in mdlUnique(Prg, M).
- 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 single 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 $\overline{\mathtt{Dlt}}$ containing exactly those deltas of M activated by π, respecting the order among deltas specified in the configuration knowledge of M.

For technical reasons we extend the set of products of each VM M with a dedicated auxiliary product, denoted $\perp$, identifying the unique part of M.

Definition 7 (*Extended Product*)**.** An *extended product xc* of a VM M is either a product π of M or the symbol $\perp$ not used for any other product.

We define notation for extracting the meaning of ground (Definition 1) dependencies in the Module Core Part of a given VM of a given program. In the following, we use δ to range over dependencies.

[7] Therefore, because of sanity condition (7), export/import clauses no longer contain product names.

[8] This is always possible, because configuration definitions and closed product definitions cannot depend on open product definitions by sanity conditions (8a)(ii), (8b)(ii), and (8c)(ii).

[9] Because of sanity condition (9)(v).

Definition 8 (*Ground Dependency Meaning*)**.** Given a program Prg, a VM M of Prg, a ground dependency δ on M'.N occurring in M, and an extended product xc of M.

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

If all dependencies in the core part MdlC of M are ground define $\Downarrow(\texttt{Prg},\texttt{M},\texttt{MdlC},xc)=\{\Downarrow(\texttt{Prg},\texttt{M},\delta,xc)\mid\delta \text{ is a dependency in } \texttt{MdlC}\}$.

Flattening a program Prg may require to generate more than one variant for each of its VMs. The flattening process generates new names for the generated Vf-ABS modules implementing the required variants and translates the dependencies occurring in Prg into uses of (i.e., with-free dependencies on) the generated names. Next we define notation 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 9 (*New Module Name, Dependency Translation*)**.** Given a program Prg, a VM M of Prg, let xc be either $\bot$ or a product π of M. We denote with $\Uparrow(\texttt{M},xc)$ the name of the module that implements the unique part of the variants of M when $xc=\bot$, otherwise, the name of the module that implements the non-unique part of the variant of M for product xc. Moreover, for any ground dependency δ on M'.N:

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

If all dependencies in the core part MdlC of M are ground 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 $\Uparrow(\texttt{Prg},\texttt{M},\delta,xc)$.

7.2. Flattening

The following definition formalizes the application of an ordered delta sequence $\overline{\texttt{Dlt}}$ (the deltas activated by a product π of a VM M) to a sequence $\overline{\texttt{Defn}}$ of interface/class definitions (the non-unique class/interface definitions in the module core part MdlC of M). This is the standard semantics of delta application (Schaefer et al., 2010).

Definition 10 (*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 Appendix A.

According to the definition above, application of a delta succeeds under the following conditions: (i) each class/interface to be removed or modified is present in the module in focus; (ii) each attribute to be removed or modified is present (with identical header) in the class/interface in focus; (iii) each class, method or field to be added is not present already; (iv) when adding/removing interface references from $\overline{\texttt{IR}}$ in class C implements $\overline{\texttt{IR}}$ ... and interface I extends $\overline{\texttt{IR}}$ ..., the sequence $\overline{\texttt{IR}}$ is treated as a set: first interface references are added to $\overline{\texttt{IR}}$ (this operation succeeds even if some of the added interface references are already present in $\overline{\texttt{IR}}$) yielding $\overline{\texttt{IR}}'$—*only then* interface references are removed (this operation succeeds even when some of the removed interface references are not present in $\overline{\texttt{IR}}'$).

Let π be a product of M. We define a mapping $\sigma=\mathsf{genP}(\texttt{Prg},\texttt{M},\pi)$ from open product names defined by M to their corresponding set of features. Given a sequence of interface/class definitions $\overline{\texttt{Defn}}$ and such a mapping σ, we denote with $\sigma(\overline{\texttt{Defn}})$ the definitions obtained from $\overline{\texttt{Defn}}$ by replacing each occurrence of a open product name with its associated set of features.[10]

We are ready to define the rules that flatten a VM to produce either a Vf-ABS module containing its unique class/interfaces, or a Vf-ABS module containing the non-unique class/interfaces generated for a given product:

Definition 11 (*Local Flattening of VM Relative to an Extended Product*)**.** Let M be the name of a VM of Prg, xc an extended product of M. The *local flattening of* M *relative to* xc is the Vf-ABS module Mdl such that the judgment $\texttt{M}\xrightarrow{\texttt{Prg},xc}\ D,\texttt{Mdl}$ (defined by rules LF:VM$_\bot$ and LF:VM$_{\not\bot}$ below) holds, where: (i) Mdl is the code of a Vf-ABS module named $\Uparrow(\texttt{M},xc)$, which, for the case $xc=\bot$ implements the unique part of the variants of M, for the case $xc=\pi$ implements the non-unique part of the variant of M for product π; (ii) D is the set of pairs (module, extended product) identifying the set of variants from which $\Uparrow(\texttt{M},xc)$ depends on the rules given in Box I.

Rule LF:VM$_\bot$ generates a module implementing the unique part of the variants of a given VM 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 Vf-ABS modules that need to be generated) and a new Vf-ABS module named $\Uparrow(\texttt{M},\bot)$ that: (i) exports everything; (ii) imports from all Vf-ABS modules identified in D; (iii) contains the unique classes/interfaces of the original VM, where all syntactic dependencies are translated according to $\Uparrow$.

Rule LF:VM$_{\not\bot}$ generates a Vf-ABS module implementing the non-unique part of the variant of the VM M for a 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 Definition 10 before being integrated in the resulting module.

Example 2 (*Local Flattening*)**.** Consider the following VM M, which declares two interfaces, only one of which is unique.

```
module M;
export J;
features F with true;
unique interface I {}
interface J extends I {}
delta D; modifies interface J { adds Unit m(); }
delta D when F;
```

Rule LF:VM$_\bot$ generates the module implementing the unique part of the VM with added import and export clauses.

```
module M;
export *;
import * from M_F;
interface I {}
```

Rule LF:VM$_{\not\bot}$ for $\pi=\{\texttt{F}\}$ generates the following module:

```
module M_F;
export J;
import * from M;
interface J { Unit m(); }
```

[10] Since we are considering programs that satisfy the sanity conditions and are in normal form, this set of features is a product.

LF:VM$_\perp$

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

LF:VM$_\not\perp$

$$\frac{\begin{array}{c}\mathsf{mdlNotUnique}(\mathtt{Prg},\mathtt{M}) = \overline{\mathtt{Defn}} \quad \mathsf{mdlDelta}(\mathtt{Prg},\mathtt{M},\pi) = \overline{\mathtt{Dlt}} \\ (\overline{\mathtt{Dlt}},\ \overline{\mathtt{Defn}}) \rightarrow^* \overline{\mathtt{Defn}'} \quad \sigma = \mathsf{genP}(\mathtt{Prg},\mathtt{M},\pi) \\ \sigma(\overline{\mathtt{Defn}'}) = \overline{\mathtt{Defn}''} \quad \Downarrow(\mathtt{Prg},\mathtt{M},\overline{\mathtt{Defn}''},\perp) = D = \{(\mathtt{M}_i, xc_i) \mid i \in I\}\end{array}}{\mathtt{M} \xrightarrow{\mathtt{Prg},\pi} D, \mathtt{module}\ \Uparrow(\mathtt{M},\pi);\ \mathtt{export}\ *;\ \overline{\mathtt{import}\ *\ \mathtt{from}\ \Uparrow(\mathtt{M}_i, xc_i);}\ \Uparrow(\mathtt{Prg},\mathtt{M},\overline{\mathtt{Defn}''},\pi)}$$

Box I.

CF:VM1

$$\frac{xc \text{ is an extended product of } \mathtt{M} \quad \mathtt{M} \xrightarrow{\mathtt{Prg},xc} D, \mathtt{Mdl} \quad A = \{(\mathtt{M}, xc)\}}{\varepsilon, \emptyset, \{(\mathtt{M}, xc)\} \xrightarrow{\mathtt{Prg}} \mathtt{Mdl}, A, (D \setminus A)}$$

CF:VM2

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

Box II.

Next we define rewrite rules that, given a VM M of Prg and an extended product *xc* of M, generate the variant of M identified by *xc* as well as all variants of the VMs in Prg needed to resolve the dependencies of all the variants that arise during the rewriting.

Definition 12 (*Complete Flattening of VM Relative to an Extended Product*)**.** Let ε denote the empty program, representing the initial partial result of a complete flattening of a VM M of an ABS-VM program Prg relative to an extended product *xc*.

The two rules below define a judgment of the form $\mathtt{Prg}', A_1, D_1 \xrightarrow{\mathtt{Prg}} \mathtt{Prg}'', A_2, D_2$, where: Prg′ (either ε or a Vf-ABS program) is a partial result of complete flattening, the set A_1 identifies the already generated Vf-ABS modules, the set D_1 identifies the Vf-ABS modules that must be generated to resolve the dependencies in Prg′, the Vf-ABS program Prg″ is obtained by adding to Prg′ the code of one of the Vf-ABS modules identified by D_1, and the sets A_2, D_2 are obtained by suitably updating A_1, D_1, respectively (see rules given in Box II).

Let $\xrightarrow{\mathtt{Prg}}{}^*$ be the transitive closure of $\xrightarrow{\mathtt{Prg}}$. The *complete flattening of VM* M *of* Prg *relative to its extended product xc* is the Vf-ABS program Prg′ such that $\varepsilon, \emptyset, \{(\mathtt{M}, xc)\} \xrightarrow{\mathtt{Prg}}{}^* \mathtt{Prg}', A, \emptyset$ holds.

The sets A, D, A_1, D_1, A_2 and D_2 in the two rules above refer to dependencies in the original program Prg. Rule CF:VM1 starts with the empty ABS program, the empty set of resolved dependencies, and the singleton set $\{(\mathtt{M}, xc)\}$ of dependencies to be resolved. It adds the ABS module implementing the extended product *xc* of M and updates the dependency sets. Rule CF:VM2 extends Prg′ by adding the ABS module required by one of the dangling dependencies in D_1, and replaces the dependency sets A_1, D_1 by their updated versions A_2, D_2, respectively.

Finally, we can formalize flattening of a whole ABS-VM program.

Definition 13 (*Flattening of ABS-VM Program*)**.** The *flattening of an ABS-VM program* Prg is the complete flattening of its main module relative to $\perp$: the Vf-ABS program Prg′ such that $\varepsilon, \emptyset, \{\mathsf{mdlInit}(\mathtt{Prg}), \perp)\} \xrightarrow{\mathtt{Prg}}{}^* \mathtt{Prg}', A, \emptyset$ holds.

```
module M0;
import * from MF;
init { MF.J j = null; }

module M;
export *;
import * from MF;
interface I {}

module MF;
export J;
import * from M;
interface J { Unit m(); }
```

Fig. 8. Complete flattening of VM M0 from Example 2.

Example 3 (*Complete Flattening*)**.** Let us continue with Example 2. Consider the program Prg that consists of M and the module $\mathtt{M}_0$ given below

```
module M0;
import * from M;
init { J with {F} j = null; }
```

The complete flattening of $\mathtt{M}_0$ is given in Fig. 8.

8. Type-safety for ABS-VM programs

In this section, as in Section 7, we consider ABS-VM programs in normal form (Definition 5). We first define well-typed Vf-ABS programs (Section 8.1), then type-safe ABS-VM programs (Section 8.2).

8.1. Well-typed Vf-ABSprograms

We assume Vf-ABS programs to be sane (Section 5.2). First we set up terminology for what we call the basic types related to Vf-ABS programs.

Definition 14 (*Basic Type*)**.** A *basic type* BT of a Vf-ABS program is one of the following:

```
module M;
export I, J, C;
interface I { Int getK(); }
interface J { Unit inc(); }
class C implements I, J {
  Int k= 0;
  Int getK() { return k; }
  Unit inc() { k = k+1; }
}
```

$$\begin{aligned}
\mathtt{PrgS(M)(export)} &= \mathtt{M.I,\ M.J,\ M.C} \qquad \mathtt{PrgS(M)(import)} = \emptyset \\
\mathtt{PrgS(I)(super)} &= \emptyset \qquad \mathtt{PrgS(J)(super)} = \emptyset \qquad \mathtt{PrgS(C)(super)} = \{\mathtt{M.I}, \mathtt{M.J}\} \\
\mathtt{PrgS(I)(getK)} &= \mathtt{()} \to \mathtt{Int} \qquad \mathtt{PrgS(J)(inc)} = \mathtt{()} \to \mathtt{Unit} \\
\mathtt{PrgS(C)(k)} &= \mathtt{Int} \qquad \mathtt{PrgS(C)(getK)} = \mathtt{()} \to \mathtt{Int} \qquad \mathtt{PrgS(C)(inc)} = \mathtt{()} \to \mathtt{Unit}
\end{aligned}$$

Fig. 9. A program and its signature.

1. A *variable basic type* VBT: a primitive type PT or a qualified interface name of the shape M.I (so VBT is a strict subset of variable types VT, see Fig. 6).
2. An *expression basic type* EBT: a variable basic type or a qualified class name of the form M.C.
3. A *method basic type* MBT of the form $(\ \overline{\mathtt{VBT}}\ ) \to \mathtt{VBT}$, where $\overline{\mathtt{VBT}} = \mathtt{VBT}_1,\ldots,\mathtt{VBT}_n$ ($n \geq 0$) are the types of the formal parameters and VBT is the return type.
4. A *reference basic type* RBT: a qualified class/interface name of the form M.N.

The next definition provides signatures to represent the structure of classes, interfaces, modules, and programs without method implementations. The subsequent definitions introduce the subtyping relation induced by a program signature and the notion of a well-formed program signature, respectively.

Definition 15 (*Class/interface/module/program signature*)**.**

1. A *class signature* consists of the class name, a mapping from the keyword super to the set of qualified names of the implemented interfaces, a mapping from each field name to its variable basic type, and a mapping from each method name to its method basic type. Class signatures are ranged over by CS. We write sig(CD) to denote the signature of class declaration CD.
2. An *interface signature* consists of the interface name, a mapping from the keyword super to the set of qualified names of the extended interfaces and a mapping from each method name to its method basic type. Interface signatures are ranged over by IS. We write sig(ID) to denote the signature of the interface declaration ID.
3. A *module signature* consists of the module name M, a mapping from the keyword export to its trade clause, a mapping from the keyword import to the set of all imported pairs $(\mathtt{tC}_i, \mathtt{M}_i)$ with $n \geq 0$ and $\mathtt{M}_i \neq \mathtt{M}$ $(1 \leq i \leq n)$, and a mapping from each class/interface name to its class/interface signature. Module signatures are ranged over by MdlS. We write sig(Mdl) to denote the signature of the module Mdl.
4. A *program signature* maps each module name to its module signature. Program signatures are ranged over by PrgS. We write sig(Prg) to denote the signature of the program Prg.

A program signature sig(Prg) can be computed by inspection of Prg, specifically, ID and sig(ID) are isomorphic.

Example 4 (*Program Signature*)**.** Fig. 9 shows a program Prg and its signature sig(Prg) = PrgS.

Definition 16 (*Subtyping Relation Induced by Program Signature*)**.** The *subtyping relation induced by program signature* PrgS, denoted by $<:_{\mathtt{PrgS}}$, is the transitive closure of the union of: (i) the identity relation on primitive types and qualified class/interface names defined in Prg, (ii) the implements-relation (defined by class implements clauses), (iii) the immediate extends-relation (defined by the interface extends clauses). When PrgS is clear from the context we write simply $<:$.

Definition 17 (*Well-Formed Program Signature*)**.** A program signature PrgS is *well-formed* iff:

1. for every module name M appearing anywhere in PrgS we have $\mathtt{M} \in \mathsf{dom}(\mathtt{PrgS})$;
2. for every qualified class/interface name M.N appearing anywhere in PrgS we have $\mathtt{N} \in \mathsf{dom}(\mathtt{PrgS(M)})$;
3. every traded name appearing in PrgS(M)(export) occurs in dom(PrgS(M));
4. if PrgS(M)(import) contains a pair (tC,M′) then $\mathtt{M}' \neq \mathtt{M}$ and all traded names in tC occur in PrgS(M′)(export);
5. the transitive closure of the immediate extends-relation is acyclic;
6. for all modules $\mathtt{M}, \mathtt{M}_i \in \mathsf{dom}(\mathtt{PrgS})$, classes/interfaces $\mathtt{N}, \mathtt{N}_i \in \mathsf{dom}(\mathtt{PrgS(M)})$, if $\mathtt{M.N} <: \mathtt{M}_1\mathtt{.N}_1$ and $\mathtt{M.N} <: \mathtt{M}_2\mathtt{.N}_2$, then for each field $\mathtt{a} \in \mathsf{dom}(\mathtt{PrgS}(\mathtt{M}_1)(\mathtt{N}_1)) \cap \mathsf{dom}(\mathtt{PrgS}(\mathtt{M}_2)(\mathtt{N}_2))$, it holds that $\mathtt{PrgS}(\mathtt{M}_1)(\mathtt{N}_1)(\mathtt{a}) = \mathtt{PrgS}(\mathtt{M}_2)(\mathtt{N}_2)(\mathtt{a})$.

The signature in Example 4 is well-formed. The typing rules for Vf-ABS are straightforward and follow a standard pattern for simple OO programs with modules.

Definition 18 (*Well-Typed Vf-ABS Module and Program*)**.**

1. Let PrgS be the well-formed signature of a Vf-ABS program. A module Mdl such that PrgS(name(Mdl)) = sig(Mdl) is *well-typed* iff the judgment $\mathtt{PrgS} \vdash \mathtt{Mdl}$ can be derived by the rules given in Appendix B.
2. A Vf-ABS program Prg is *well-typed* iff it is sane and the judgment $\vdash \mathtt{Prg}$ can be derived by the following rule:

$$\textsc{T:Prg}\ \frac{\mathtt{sig(Prg)}\text{ is well-formed} \qquad \mathtt{sig(Prg)} \vdash \mathtt{Mdl}\text{, for all } \mathtt{Mdl} \in \mathtt{Prg}}{\vdash \mathtt{Prg}}$$

8.2. Type-safety

We begin with an auxiliary definition based on local flattening (Definition 11).

Definition 19 (*Overall Flattening of VM and Program*)**.** Let Prg be an ABS-VM program and M be a VM of Prg.

1. The *overall flattening* of M is the Vf-ABS program consisting of all Vf-ABS modules Mdl, where Mdl is the local flattening of M relative to one of its extended products.
2. The *overall flattening* of Prg is the Vf-ABS program consisting of all Vf-ABS modules Mdl occurring in the overall flattening of some VM of Prg.

Because of sanity conditions 1, 8, and 9 (Section 5.2) we can observe that if the overall flattening of Prg succeeds then for all its VMs Mdl, the complete flattening of Mdl relative to each of its extended products succeeds (and its modules are a subset of the modules of the overall flattening of Prg). In contrast, if the flattening of Prg succeeds this does *not* imply that the overall flattening of Prg succeeds (the local flattening of some VM relative to one of its products not required by the main module may fail).

We can now define type-safe ABS-VM programs in terms of well-typed Vf-ABS programs (Definition 18).

Definition 20 (*Type-Safe ABS-VM Program*)**.** The ABS-VM program Prg is *type-safe* iff its overall flattening (i) succeeds and (ii) produces a well-typed Vf-ABS program.

We observe that when Prg is type-safe, for all its VMs Mdl, the complete flattening of Mdl relative to each of its extended products generates a well-typed Vf-ABS program. Moreover, when a type-safe ABS-VM Program $\texttt{Prg}_1$ is modified by only changing its main module, then type-safety of the modified program $\texttt{Prg}_2$ can be checked simply by typing the local flattening Mdl′ relative to $\bot$ (of the modified main module) in the context of the program signature of the flattening $\texttt{Prg}_2'$ of $\texttt{Prg}_2$. This corresponds to proving the judgment $\mathsf{sig}(\texttt{Prg}_2') \vdash \texttt{Mdl'}$ (Definition 18). Likewise, adding a product π to a VM $\texttt{Mdl}_1$ of a type-safe program Prg, type-safety of the modified program $\texttt{Prg}_2$ can be checked simply by typing the local flattening $\texttt{Mdl}_2'$ of the modified VM $\texttt{Mdl}_2$ relative to π in the context of the program signature of the compete flattening $\texttt{Prg}_2'$ of $\texttt{Mdl}_2$ relative to π. This corresponds to proving the judgment $\mathsf{sig}(\texttt{Prg}_2') \vdash \texttt{Mdl}_2'$ (Definition 18).

Checking type-safety of a program Prg by generating its overall flattening is generally unfeasible due to the number of products of each VM that may be exponential in the number of features. In the next section we provide means for partially checking type-safety in a feasible manner.

9. Family-based checking for ABS-VM programs

ABS-VM program sanity-conditions (Section 5.2), PEV-compliance (Definition 3), inference of the maximal set of unique annotations (Section 6.2), and adherence to normal form (Definition 5) are checkable in a family-based way. We present three family-based analyses for ABS-VM programs in normal form (see Definition 5):

- Type uniformity (Section 9.1) checks whether type declarations are uniform across the different variants of each VM.
- Pre-typing (Section 9.2) checks whether the module core part and the deltas are consistent with the uniform typing information.
- Applicability consistency (Section 9.3) checks whether overall flattening succeeds.

Type uniformity helps developing and maintaining ABS-VM programs, as in any part of a program all occurrences of a given field/method always have the same type (Damiani and Lienhardt, 2016).

Pre-typing assumes type-uniformity. Like type uniformity, it does not use any knowledge about valid feature combinations (it does not use module headers). So it does not guarantee that the overall flattening of a program is well-typed (some class/interface/method/field declared in some VM may not be present in its local flattening relative to one of its products). However, pre-typing guarantees that if each module in the overall flattening has its dependencies fulfilled (i.e., each used module/class/interface/attribute is present and, for each required subtyping relation, suitable extends/implements clauses that define it are present), then the overall flattening is well-typed. Therefore, to ensure that a pre-typed ABS-VM program is type-safe (Definition 20) it is sufficient that: (i) its overall flattening succeeds and (ii) it has all its dependencies satisfied.

Applicability consistency checks condition (i). Therefore, the three family-based analyses presented in this section are able to guarantee type-safety modulo the fact that all the dependencies in the overall flattening are satisfied.[11]

9.1. Type uniformity

Intuitively, an ABS-VM program is *type uniform* when its extends-relation is acyclic and field/method types and subtyping relations are consistent across program declarations: all declarations of the same field/method in the same class/interface must have the same type.

To formalize the notion of type uniformity we introduce three auxiliary definitions that introduce types (a superset of the basic types of Definition 14), a more liberal version of program signatures (Definition 15), and a generalization of the subtype relation described by a program signature (Definition 16).

Definition 21 (*Types and Type Stripping*)**.** A *type* T is one of:

1. An *expression type* ET: a variable type VT (i.e., an interface reference or a basic type, see Fig. 6), a qualified class name of the form M.C, or M.C with KE.
2. A *method type* MT: it has the form $(\ \overline{\texttt{VT}}\ ) \to \texttt{VT}$, where $\overline{\texttt{VT}} = \texttt{VT}_1, \ldots, \texttt{VT}_n$ ($n \geq 0$) are the types of the formal parameters, and VT is the return type.
3. A *reference type* RT: an interface reference IR, a qualified class name of the form M.C, or M.C with KE.

The *stripping of a type* T, denoted by strip(T), is the basic type (Definition 14) obtained from T by dropping the with-clauses occurring it.

Definition 22 (*Family class/interface/module/program signature*)**.**

1. A *family class signature*, ranged over by FCS, aggregates the information of all signatures of a class in all variants.
   - It contains the class name.
   - It associates the keyword super with the set R of the interface references that may be supertypes of the class. Let M be the VM of the class. Then the set R contains all interface references
     - that are listed in the implements-clause of any declaration of the class in the core part of M or in a delta of M,
     - that are added to the implements-clause of the class by a modifies-operation on the class in a delta of M.

[11] We are working on a family-based analysis for performing this latter check as well (Section 12).

- It associates each field/method name a, which may be an attribute of the class, with the non-empty set of the variable/method types of a. Let M be the VM of the class. Then the domain of the mapping contains the names of the fields/methods
  - that are defined in a class declaration occurring in the core part of M or in a delta of M,
  - that are subject of an attribute operation in a `modifies`-operation on the class in a delta of M.

2. A *family interface signature*, ranged over by FIS, aggregates the information of all signatures of an interface in all variants.
   - It contains the interface name.
   - It associates the keyword `super` with the set R of interface references that may be supertypes of the interface. Let M be the VM of the interface. Then the set R contains all the interface references
     - that are listed in the `extends`-clause of any definition of the interface in the core part of M or in a delta of M,
     - that are added to the `extends`-clause of the interface by a `modifies`-operation on the interface in a delta of M.
   - It associates each method name m, which may be a method of the interface, with the non-empty set of the method types of m. Let M be the VM of the interface. Then the domain of the mapping contains the names of the methods
     - that are defined in an interface declaration occurring in the core part of M or in a delta of M,
     - that are subject of a header operation in a `modifies`-operation on the interface in a delta of M.
3. A *family module signature*, ranged over by FMdlS, aggregates the information of all signatures of a module in all variants.
   - It contains the module name M.
   - It associates the keyword `export` with its exported trade clause `tC`.
   - It associates the keyword `import` with the set of all imported pairs $(\texttt{tC}_i, \texttt{M}_i)$, where $n \geq 0$ and $\texttt{M}_i \neq \texttt{M}$ $(1 \leq i \leq n)$.
   - It associates each class/interface name (defined in the core part of M and/or subject of a class/interface operation in a delta of M) with its family class/interface signature.
   - It associates the name of each open product (defined in M) with its definition.
   - It optionally contains a feature model, denoted with FMdlS.FM.

   We write fsig(`Mdl`) to denote the family signature of `Mdl`.
4. A *family program signature*, ranged over by FPrgS, aggregates the information of all signatures of a program in all variants. It associates each module name M with its family module signature. We write fsig(`Prg`) to denote the family signature of program `Prg`.

It is obvious that fsig(`Prg`) can be computed by inspection of `Prg`.

Definition 23 (*Subtyping Relation Induced by Family Program Signature*)**.** The *subtyping relation induced by a family program signature* FPrgS, denoted by $<:_{\texttt{FPrgS}}$, is the transitive closure of the union of the following two relations:

1. The identity relation on primitive types and on the qualified class/interface names in dom(FPrgS);
2. $\{(\texttt{M.N}, \text{strip}(\texttt{IR})) \mid \texttt{M} \in \text{dom}(\texttt{FPrgS}), \texttt{N} \in \text{dom}(\texttt{FPrgS(M)}), \texttt{IR} \in \texttt{FPrgS(M)(N)(super)}\}$.

When FPrgS is clear from the context, we simply write $<:$ instead of $<:_{\texttt{FPrgS}}$.

It is worth observing that the relation in Definition 23. 2 might be cyclic.

We can now formalize type uniformity as follows.

Definition 24 (*Type Uniformity*)**.** A family program signature FPrgS is type uniform if the following conditions hold:

1. For every VM name M appearing anywhere in FPrgS, we have $\texttt{M} \in \text{dom}(\texttt{FPrgS})$.
2. For every qualified class/interface name M.N appearing anywhere in FPrgS, we have $\texttt{N} \in \text{dom}(\texttt{FPrgS(M)})$;
3. For every VM name $\texttt{M} \in \text{dom}(\texttt{FPrgS})$, every class/interface name $\texttt{N} \in \text{dom}(\texttt{FPrgS(M)(N)})$, and every attribute name $\texttt{a} \in \text{dom}(\texttt{FPrgS(M)})$ the set FPrgS(M)(a) is a singleton.
4. For every traded name `tN` appearing in FPrgS(M)(`export`) that occurs in dom(FPrgS(M)), we have $\texttt{N} \in \text{dom}(\texttt{FPrgS(M)})$.
5. If PrgS(M)(`import`) contains a pair (`tC`,M') then $\texttt{M'} \neq \texttt{M}$ and all traded names in `tC` occur in PrgS(M')(`export`).
6. The relation given in item 2 of Definition 23 is acyclic.
7. For all VMs $\texttt{M}, \texttt{M}_1, \texttt{M}_2 \in \text{dom}(\texttt{FPrgS})$, classes/interfaces $\texttt{N} \in \text{dom}(\texttt{FPrgS(M)})$, interfaces $\texttt{I}_i \in \text{dom}(\texttt{FPrgS}(\texttt{M}_i))$ $(i \in \{1, 2\})$: if $\texttt{M.N} <: \texttt{M}_1\texttt{.I}_1$ and $\texttt{M.N} <: \texttt{M}_2\texttt{.I}_2$ then for each attribute $\texttt{a} \in \text{dom}(\texttt{FPrgS}(\texttt{M}_1)(\texttt{I}_1)) \cap \text{dom}(\texttt{FPrgS}(\texttt{M}_2)(\texttt{I}_2))$ we have $\texttt{FPrgS}(\texttt{M}_1)(\texttt{I}_1)(\texttt{a}) = \texttt{FPrgS}(\texttt{M}_2)(\texttt{I}_2)(\texttt{a})$.

An ABS-VM program PrgS is type uniform iff fsig(PrgS) is type uniform.

Type uniformity of a program can be checked by inspecting its family signature.

Example 5 (*Type Uniformity*)**.** Consider the program `Prg` in Fig. 10. The code is type uniform and the program signature $\text{fsig}(\texttt{Prg}) = \texttt{FPrgS}$ is such that:

$$\texttt{FPrgS(M)(export)} = \emptyset \quad \texttt{FPrgS(M)(import)} = \emptyset$$
$$\texttt{FPrgS(I)(super)} = \emptyset \quad \texttt{FPrgS(J)(super)} = \{\texttt{M.I}\}$$
$$\texttt{FPrgS(J)(m)} = \{\texttt{()} \to \texttt{Unit}\}$$

The program looses type uniformity, for example, if one adds the code below to module M.
This would imply $\texttt{FPrgS(J)(m)} = \{\texttt{()} \to \texttt{Unit}, \texttt{(Int)} \to \texttt{Unit}\}$, which breaks type uniformity (third condition).

9.2. Pre-typing

We say an ABS-VM program `Prg` is *pre-typed* to mean that it is type uniform and all method declarations in the core/delta parts of VMs type check with respect to fsig(`Prg`). This is formalized by the following by a quite straightforward adaptation of the definition well-typed Vf-ABS program (Definition 18).

Definition 25 (*Pre-Typed ABS-VM Program*)**.**

1. Let FPrgS be the program signature of a type uniform Vf-ABS program. A VM `Mdl` such that $\texttt{FPrgS(name(M))} = \text{fsig}(\texttt{Mdl})$ is *pre-typed* iff the judgment $\texttt{FPrgS} \Vdash \texttt{Mdl}$ can be derived by the rules given in Appendix C.

```
delta D3; modifies interface J { adds Unit m(Int i); }
delta D3 when F2;
```

```
module M;
features F1, F2 with F1 <-> !F2;
unique interface I {}
interface J extends I {}
delta D1; modifies interface J { adds Unit m(); }
delta D2; modifies interface J { modifies Unit m(); }
delta D1 when F1;
delta D2 when F1 after D1;
```

Fig. 10. A type uniform program.

2. An ABS-VM program Prg is *pre-typed* iff the judgment $\Vdash$ Prg can be derived by the following rule:

$$\text{PT:PRG}\quad \frac{\text{fsig(Prg) is type uniform} \qquad \text{fsig(Prg)} \Vdash \text{Mdl, for all Mdl} \in \text{Prg}}{\Vdash \text{Prg}}$$

Remark 3 (*Early Error Detection*)**.** PEV-compliance (Definition 3), normal form adherence (Definition 5), type uniformity (Definition 24), and pre-typing (Definition 25) can be relaxed by extending them to ABS-VM programs that do *not* satisfy the SAT sanity conditions (final bullet in Section 5.2). These relaxed properties, called SAT-free X (where X $\in$ {PEV-compliance, normal form, type uniformity, pre-typing}) can be checked by inspection without a SAT solver. It is also obvious that, if an ABS-VM program satisfies property SAT-free X plus the SAT sanity conditions, then it satisfies property X. Therefore, ABS-VM program developers may frequently trigger an inexpensive check of the SAT-free X properties (an IDE can perform them in the background, providing immediate detection of the associated errors), while being more careful in triggering the (potentially more time consuming) checks involving a SAT solver.

9.3. Applicability consistency for normal form ABS-vm programs

In this section we assume each feature f is qualified by the name of the VM M, where its feature model is defined (denoted M.f) and each delta name D is qualified by the name of its VM (denoted M.D).

Let *getFm*(Prg, M) denote the feature model formula of VM M (given in its module header), let *getFm*(Prg) denote the conjunction of all feature model formulas $\bigwedge_{\text{M}\in\text{dom(PrgS)}} \mathit{getFm}(\text{Prg}, \text{M})$, and let *getAct*(Prg, M.D) denote the activation condition formula of the delta with name D defined in VM M (given in its configuration knowledge).

Let $\mathcal{I}$ be an assignment from feature variables occurring in the feature models of Prg to Boolean values such that all feature formulas of Prg are satisfied. Then, for each M and such $\mathcal{I}$, this defines the product $\pi^{\mathcal{I}}_{\text{M}} = \{\text{M}.f \mid f \text{ is a feature of M and } \mathcal{I}(\text{M}.f) = \textbf{true}\}$. We define the function *getFmAct* mapping each program Prg to a formula over qualified feature names and qualified delta names (of Prg).

$$\begin{aligned} \mathit{getFmAct}(\text{Prg}) &= \textstyle\bigwedge_{\text{M}\in} \text{dom(PrgS)} \mathit{getFmAct}(\text{Prg}, \text{M}) \\ \mathit{getFmAct}(\text{Prg}, \text{M}) &= \mathit{getFm}(\text{Prg}, \text{M}) \wedge \\ &\quad \Big(\textstyle\bigwedge_{\text{D is a delta of M}} \text{M.D} \Leftrightarrow \mathit{getAct}(\text{Prg}, \text{M.D})\Big) \end{aligned}$$

We note that by construction for each assignment $\mathcal{I}$: formula *getFmAct*(Prg) is satisfied by $\mathcal{I}$ iff *getFm*(Prg) is satisfied by $\mathcal{I}$ and $\mathcal{I}(\text{M.D}) = \mathcal{I}(\mathit{getAct}(\text{Prg}, \text{M.D}))$. Hence, $\mathcal{I}(\text{M.D}) = \textbf{true}$ if and only if the delta M.D is activated by the product $\pi^{\mathcal{I}}_{\text{M}}$.

Let Q range over fully qualified class/interface names M.N and fully qualified attribute names M.N.a. Let FMdlS(M.N) be short for FMdlS(M)(N) and let FMdlS(M.N.a) be short for FMdlS(M)(N)(a). The *deep domain* of a module M such that fsig(M)=FMdlS is

$$\begin{aligned} \text{ddom(M)} = \{&\text{M.N} \mid \text{FMdlS(M.N) is defined}\} \\ &\cup \{\text{M.N.a} \mid \text{FMdlS(M.N.a) is defined}\}\ . \end{aligned}$$

The *deep domain* of a program Prg is $\text{ddom(Prg)} = \bigcup_{\text{M}\in\text{dom(PrgS)}} \text{ddom(M)}$.

Let Φ denote a Boolean formula over qualified feature names and qualified delta names of a given VM M of Prg and let KE be a product of M. Then *eval*(KE, Φ, Prg) denotes the Boolean value of the formula Φ under the assignment $\mathcal{I}$ such that:

$$\mathcal{I}(x) = \begin{cases} \textbf{true} & \text{if } x \text{ is a feature in KE} \\ \textbf{false} & \text{if } x \text{ is a feature not in KE} \\ \mathcal{I}(\mathit{getAct}(\text{Prg}, x)) & \text{if } x \text{ is a delta name} \end{cases}$$

Definition 26 (*Applicability Consistency*)**.** Let Prg be a normal form ABS-VM program and let the judgment $\vdash$ Prg : Θ, Φ be derivable with the rules given in Appendix D, where: Φ (called *applicability constraint*) is as above, and Θ (called *declaration presence mapping*) is a mapping from ddom(Prg) to propositional formulas over features and delta names. We say that Prg is *applicability consistent* to mean that the formula $\mathit{getFmAct}(\text{Prg}) \Rightarrow \Phi$ is valid.

During product generation the selected deltas must be applicable in the given order, otherwise generation would not succeed (see explanation immediately after Definition 10).

The rules in Appendix D build the applicability constraint and the declaration presence mapping of the normal form ABS-VM program Prg by parsing it from the left to right. Consistent with Section 5.1, for each VM we assume the list of deltas in Prg being a total order compatible with the configuration knowledge CK: hence, this analysis assumes deltas are applied in the order of their occurrence in Prg. At any stage t during the parsing process, the applicability constraint collects the information on the applicability of deltas until t. This constraint is defined simultaneously with the declaration presence mapping that collects for all possible subjects of delta operations the information on their presence at stage t. The applicability check succeeds for deltas applied until t, whenever the applicability constraint holds for all products.

Theorem 2 (*Soundness and Completeness of Applicability Consistency*)**.** *Let* Prg *be a normal form ABS-VM program such that* $\vdash$ Prg : Θ, Φ *holds.*

1. *The overall flattening of* Prg *succeeds iff* Prg *is applicability consistent.*
2. *Let* Prg *be applicability consistent. For every product* π *of* M*:*
   (a) *eval*(π, Θ(M.N), Prg) = **true** *iff the declaration of class/interface* N *occurs in the flattening of* M *(relative to* π *if* M.N *is not unique, relative to* $\perp$ *otherwise).*

(b) For all $M.N.a \in \mathrm{dom}(\Theta)$ it holds that: $eval(\pi, \Theta(M.N.a), \texttt{Prg}) = \textbf{true}$ iff the declaration of N occurs in the flattening of M (relative to π if $M.N$ is not unique, relative to $\perp$ otherwise) and contains the declaration of the field/method a.

Proof. The proof is by rule [A:Prg] in Appendix D and Lemma 9 in Appendix E. □

It is easy to see that using Definition 3. 1 we can strengthen the theorem to state that for all *unique* classes/interfaces the formulas collected in Θ are always **true**.

10. Integration into the ABS tool chain

We implemented the VM concept as part of the ABS compiler tool chain with the exception of open product definitions, currently under development. The implementation, together with all case studies and test cases, is available as an open source extension of the ABS compiler.[12] The `README` file 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 (Section 7.2) produces variability-free ABS code, keeping ABS code generation and semantic analysis (type checking of single variants) as is. The ABS parser's grammar is extended with the constructs described in Section 5. As expected, ABS's existing delta application mechanism, including `original` calls, could be fully reused.

Several sanity conditions, specifically those concerned with `export` and `import` clauses, boil down to simple checks once normal form is established. The SAT sanity conditions are solved using the `Choco`[13] solver, which is included in the ABS tool chain and used there already for the μTVL (Clarke et al., 2010) feature modeling language of the global product line system of ABS. Pre-typing requires to have a version of the type checker that uses the pre-computed signatures, instead of resolving type names with the class table. The error reporting mechanism is reused from the standard type checker.

The VM tool chain implementation has two aspects, several helper classes, and consists of the following components: (i) Sanity condition checking (Section 5.2) with error reporting in case a condition is violated (current implementation has limitations); (ii) PEV-compliance checking (Section 6) with error reporting in case the PEV is violated; (iii) flattening (Section 7.2); (iv) adjustment of the feature model (needed, because VMs use a simpler feature modeling language than ABS's μTVL Clarke et al., 2010).

SAT-free PEV-compliance and flattening are performed as part of the standard workflow of the compiler. For backwards compatibility, pre-typing and the applicability check are implemented as optional commands rather than mandatory steps during compilation, analogous to other static analyses for product lines in ABS such as Damiani et al. (2017b). There are two command line instructions to initiate compilation: "absc varcheck <files>" performs all checks, while "absc varcheck–nosat <files>" skips all steps requiring SAT solving: it only checks the SAT-free sanity conditions and performs pre-typing.

Delta operations on `extends` and `implements` are not supported by ABS yet, and thus, consequently, neither by our VM extension of ABS.

[12] The code is available under https://github.com/Edkamb/abstools/tree/variable_mod. A runnable VM is available under https://doi.org/10.5281/zenodo.4926115.

[13] https://choco-solver.org.

11. Evaluation

11.1. Research questions

Qualitative and quantitative aspects of VMs are evaluated along the following *research questions*:

RQ 1 Does using ABS-VM provide advantages in terms of reduced implementation effort and code readability for existing models, where variant interoperability is implemented by relying on duplicating code (possibly based on programming constructs for intra-code reuse to reduce code duplication, see Section 3) or on external tools?

RQ 2 Does using ABS-VM provide advantages in terms of reduced model size compared to Delta-ABS and Delta-Trait-ABS?

RQ 3 Does performing the family-based static normal form, type uniformity, applicability and pre-typing checks[14] (described in Section 9) require less time than *overall* flattening[15] (according to Definition 19) on an ABS-VM program?

RQ 4 Does performing the family-based static normal form, type uniformity, applicability and pre-typing checks require less time than flattening (according to Definition 13) on an ABS-VM program?

11.2. Experiment design

11.2.1. Experiment design and subject for *RQ 1* and *RQ 2*

To investigate the first two research questions, we use ABS-VM to refactor (possibly extended versions of) legacy models. The source code of the case studies is available at the URL given in footnote 12. The legacy models are as follows:

- The industrial `FormbaR` model of railway operations, available in two versions, see Remark 1. The first version, `FormbaR-1` (Kamburjan and Hähnle, 2016), is implemented in Delta-ABS. The second version, `FormbaR-2` (Kamburjan et al., 2018), is implemented in Delta-Trait-ABS. Both versions use one class per infrastructure element and, to achieve variant interoperability, different variants of the same infrastructure element are explicitly defined as separate classes without using DOP for implementing infrastructure elements.
- The second version of the *Weak Memory Model Family*, referred to as WMMF-2 and implemented in Delta-ABS. It is the extension of an ABS model of weak memory (Kamburjan and Hähnle, 2018), referred to as WMMF-1 and implemented in Delta-ABS.
  In sequentially consistent memory models all read- and write-accesses in a piece of code are processed in the stated order. Weakly consistent (for short: weak) memory models permit partial or complete re-ordering of memory access to increase efficiency. WMMF-1 formalizes different relaxation strategies and hardware models to enable simulation and analysis of their effects. A weak memory model in WMMF-1 is implemented as a class managing a list of memory accesses on a device. Variability, including different types of reordering (read before write, etc.), is implemented by DOP. WMMF-2 extends WMMF-1 to include two co-existing hardware devices with two different memory systems each: any of the four combinations of memory model can be

[14] Recall that checking normal form and applicability consistency is a way to ensure that all variants of all VMs can be generated, without actually generating them.

[15] The brute force approach to check whether all variants of all VMs can be generated: by producing them.

different. To this end, one needs four variants of the class (and module) modeling the memory model. This is achieved (in Delta-ABS) by manually copying the class `Memory` to four modules `MemoryInternal`n, and creating two copies of the class `DevicePair` in two modules `DeviceInternal`n.

- AISCO (Adaptive Information System for Charity Organizations) is a modular web portal supporting the business processes (information, reporting, spending, expenditure) of charity organizations.
  There are two versions of AISCO. The first, referred to as AISCO-1 (Setyautami et al., 2019), consists of an SPL is implemented in Delta-ABS. Its variability reflects differing legal and operational requirements of the supported organizations. Interoperability is not supported, hence variability is encoded using multiple copies of classes.
  The second version, referred to as AISCO-2 (Setyautami and Hähnle, 2021), is implemented in Java. It supports some of the functionalities of VMs by relying on an external tool, also written in Java: Different variants of a module are generated by the runtime structure, using the external tool on top of the Java implementation, see Section 12.4 for details. The code is used in production at https://amanah.cs.ui.ac.id/.

By refactoring the above models to ABS-VM, we illustrate different aspects in using VMs to model interoperable variants:

VMs vs. Traits. In `FormbaR-2`, implemented in Delta-Trait-ABS, traits[16] are used to reduce code duplication across class definitions implementing variants of the same infrastructure element. In the refactoring of `FormbaR-2`, written in Delta-Trait-ABS, to ABS-VM we show how the relevant parts of `FormbaR` are re-modeled using VMs.

VMs vs. Delta-oriented SPLs. In the refactoring of WMMF-2, written in Delta-ABS, to ABS-VM we show that in the latter there is no need to manually duplicate modules.

VMs vs. External tool chain. We compare AISCO-2, which uses an external mechanism implemented in Java to provide VM functionality,[17] with our re-implementation in ABS-VM.

Experiment design. **RQ 1** is addressed by manually refactoring the FormbaR-2, WMMF-2, and AISCO-2 model to ABS-VM and comparing *conceptually* the legacy implementation with the ABS-VM implementation. **RQ 2** is addressed by comparing the number of lines of code (LoC) before and after refactoring.

11.2.2. Experiment design and subject for *RQ 3* and *RQ 4*

For the remaining two research questions we perform experiments on the three models introduced in **RQ 1**, **RQ 2**. For each model we compare the times needed to complete the following tasks: checking normal form and type uniformity, checking applicability consistency, checking pre-typing, performing overall flattening, and performing flattening.

Additionally, for **RQ 3**, we run two experiments on synthetic data sets: We generate programs with random feature models and compare the time needed to check applicability consistency with the time needed to perform overall flattening (Synthetic Experiment 1 below) and we measure the time needed to check applicability consistency (Synthetic Experiment 2).

All experiments are performed on an Ubuntu 20.04 system running on a quadcore i7-8565U CPU @ 1.80 GHz with 32 GB RAM.

Synthetic Experiment 1. We generate for $n \in [1 \ldots 12]$ a module with n features, each feature associated with one delta adding one method $\mathtt{m}_n$ to the sole interface of the module. The module then declares one variable for each of the 2^n variants.

Synthetic Experiment 2. To evaluate applicability checking on non-trivial feature models we modify the above scenario: We generate random feature models and application conditions using the random formula generator by Roffe and Calderon (2021). Additionally, each delta now either adds, removes or modifies a method from the interface. We run this experiment for $n \in [1 \ldots 50]$.

11.3. Results for *RQ1*

11.3.1. `Formbar`

The partial refactoring of `FormbaR-2` to ABS-VM, referred to as `FormbaR-3`, has one VM with five features: `PoV` for *points of visibility* (of signals, etc.), `Speed` for signals that announce a speed restriction, `Signal` for general signals, `Main` for signals placed at the point, where the signaled aspect holds, and `Pre` for signals that announce a point where the signaled aspect holds. Hence, the VM contains features (`Main`, `Pre`, `Speed`, `Signal`, `PoV`) plus deltas for the five re-modeled kinds of signal.[18]

`FormbaR-2` uses traits to reduce code duplication in the implementation of the different kinds of signals. `FormbaR-3` has no need for traits: a set of operations is encapsulated in a delta and applied to a base `TrackElement` class. The feature model connects these operations explicitly with products corresponding to relevant infrastructure elements.

Results. The drawbacks of using an intra-product code reuse mechanism (like traits) to implement variants are discussed in Section 3. In consequence, `FormbaR-3` is (i) shorter than `FormbaR-2` (in terms of LoC), because in the latter there is a separate class for each kind of signal (see Section 11.4 for details). (ii) `FormbaR-3` is also more comprehensible. For once, its feature model makes constraints explicit that were only implicit in `FormbaR-2` (for example, that certain traits should not be used in a class at the same time). In addition, it declaratively connects code variability to the domain model. For example, in `FormbaR-3` the feature model expresses that a pre-signal has features `Pre` and `Signal` in a semantic manner in terms of a general `Signal` that announces its aspect `Pre`.

11.3.2. WMMF

In WMMF-2 we need potentially four different memory models. The Delta-ABS implementation requires to copy the memory model module *including all deltas* four times. Furthermore, the device module had to be copied twice. Essentially, we perform *manually* part of the VM flattening until we can rely on standard module operations.

Instead, the ABS-VM model, referred to as WMMF-3, contains one VM for `Memory` and one for `DevicePair` (i.e., pairs of memory models). While WMMF-3 has six features for `Memory` and eight for `DevicePair`, WMMF-2 has 40.

Results. VMs permit concise modeling of interoperable variants, as compared to modeling them explicitly with one product line and manual copying. In particular, the possibility to directly use products as type references removes the necessity to declare all possible products in advance and to decide upfront on the number of variants to be generated: ABS-VM does not require a known bound on the number of copies and can handle encapsulation using the VM mechanism.

[16] Traits (Schärli et al., 2003; Ducasse et al., 2006) 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. (2017a) for a presentation of the notion of traits supported by ABS.

[17] The VM concept was first developed and implemented for ABS (Damiani et al., 2021), then partially adapted and implemented in the Java runtime library (Setyautami and Hähnle, 2021).

[18] *Main signal*, *pre-signal*, *speed limiter*, *pre-speed limiters*, and *point of visibility* (Kamburjan et al., 2018).

11.3.3. AISCO

The requirements stipulate co-existence of multiple variants of the same feature, for example, different formats for financial reports. This is not supported in current SPL approaches, including Delta-ABS.

In AISCO-1, implemented in Delta-ABS, interoperability of multiple variants is not supported and this kind of variability is encoded using multiple copies of classes whose inclusion is regulated by the variability model. For example, financial reports may be for expense purposes, for income, or for general reporting. The distinction among these variants is embodied in three different, all classes implementing interface `Entity`. The creation of the classes is managed using deltas. This hampers inter-product code reuse. In the words of the AISCO developers, the disadvantages are that *"[...], it cannot be modeled that different product variants from the same product line co-exist and interoperate within one and the same application. This leads to suboptimal code reuse."* (Setyautami and Hähnle, 2021, referring to AISCO-1).

In AISCO-2, implemented in Java with the help of an external tool and supporting some aspects of VMs, different report variants are implemented by a VM containing a class `Report`. The variants of this VM correspond to the different classes defined explicitly in AISCO-1. The auxiliary concept of an `Entity` is not needed any longer.

For the ABS-VM model, referenced to as AISCO-3, the main aspects of AISCO-1/AISCO-2 were re-implemented in ABS-VM in 160 LoC: One VM with four features and five 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 framework, as in AISCO-2, that is intertwined with the generated program.[19]

Results. The AISCO-2 implementation is distributed between Java and external tools that weave interoperable variants together in the desired manner. This approach has two major drawbacks: first, in contrast to AISCO-3, the various sanity, compliance, and type checks discussed in Section 9 cannot be easily realized. Even for a concrete generated program type-safety cannot be fully statically checked, because the VM implementation mechanism of Setyautami and Hähnle (2021) relies on reflection. Second, the distribution of configuration knowledge to different places in the implementation makes maintenance harder and more error-prone than in AISCO-3.

A Java-specific limitation also present in AISCO-2 is the impossibility to remove the implementation (method, class) of a functionality, one can only override it. As the VM concept is based on DOP, this is not an issue for AISCO-3.

The AISCO case study reinforces the experimental evidence that VMs are a suitable concept to simplify *heterogeneous* applications by modeling variability and co-existence of variants within a single, coherent language.

11.4. Results for **RQ 2**

We discuss the quantitative aspects of the VM refactorings to address **RQ 2**. The AISCO-1 and AISCO-2 models are unsuitable for a quantitative comparison, as AISCO-2 is not written in ABS and AISCO-1 contains external aspects (like frontend routing).

[19] For example, routing of the frontend of the web application is located in the external tool.

11.4.1. FormbaR

Compared to `FormbaR-2`, the `FormbaR-3` VM version reduces the total number of LoC needed for the signal products from 241 to 180 (−25%). Excluding code required only for variability modeling (configuration knowledge and delta headers), the re-modeled part has 163 LoC (−33%). There are various effects that lead to a reduced number of LoC:

- `FormbaR-2` declares one interface and one class per infrastructure kind[20] In contrast, `FormbaR-3` declares a single interface `Specific`, which is then transformed by deltas concurrently with its corresponding class. This reduces the header for interface declarations, in particular, when the interface needs not be changed (for example, the interface for `PoV` contains no methods and is only declared to provide a type). This means that one uses, for example, `Specific with [PoV]` instead of `IPoV`.
- By using deltas, in `FormbaR-3` we can share not merely methods (through traits), but also fields. Additionally, using a core class, we can share the `init` block of classes which is common to most classes.
- In `FormbaR-3`, instead of using trait expressions inside class definitions (with the `uses` declaration Damiani et al., 2017a), we rely on selection of the infrastructure type at the type reference, which is external to the class.

It is worth observing that all these improvements allow more concise modeling *without* changing the underlying class model of ABS, which is specifically designed to simplify deductive verification and static analysis (Hähnle, 2013).

11.4.2. WMMF

The WMMF-3 model has 485 LoC, of which 440 LoC are the two variable modules. The WMMF-2 model 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 for each combination. Hence, in addition to the additional delta declarations, this blows up the feature model unreasonably.

11.5. Results for **RQ 3** and **RQ 4**

11.5.1. Experiments on the FormbaR, WMMF, and AISCO case study

To answer **RQ 3** and **RQ 4**, for each of the models Formbar-3, WMMF-3, and AISCO-3 (listed in Table 1), we compare:

- The time needed to perform the family-based static normal form, type uniformity,[21] applicability and pre-typing checks,
- the time needed to perform overall flattening and flattening.[22]

Family-based static checking succeeds for all three models. The results are shown in Table 2 (all times are in seconds). These experiments show for all considered case studies that checking normal form, type uniformity, and applicability is faster than

[20] One cannot remove the interfaces, because classes are not types in ABS.

[21] The time for checking type uniformity includes the time for building the family signature of the program (Definition 22).

[22] Without checking whether the generated program is well typed.

Table 1
Number of features and variants for each VM in the case studies.

Model	VM	# of features	# of products
FormbaR-3	`TrackElements`	5	5
WMMF-3	`Mem`	6	64
	`Devices`	8	255
AISCO-3	`FinancialReport`	4	12

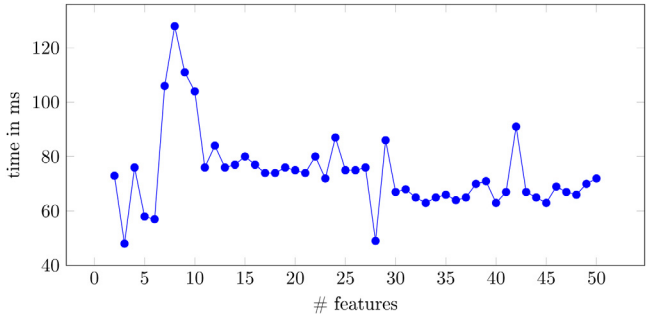

Fig. 11. Results for Synthetic Experiment 2.

performing overall flattening (answering **RQ 3**) and flattening (answering **RQ 4**). Just the time for checking pre-typing is nearly the same as the time for overall flattening in WMMF-3, while it is one other of magnitude higher than the time for overall flattening in `FormbaR`-3 and AISCO-3, and one order of magnitude higher than the time for flattening in all the three case studies.

11.5.2. Experiments on synthetic data sets

To answer **RQ 3** we perform two additional experiments on synthetic data sets (Synthetic Experiment 1 and 2, see Section 11.2.2). The results of Synthetic Experiment 1 are shown in Table 3 (all times are in seconds). Family-based static checks are run on programs with an *empty* main block, to subtract artifacts from the exponentially growing main block. In Synthetic Experiment 1 overall flattening shows exponential growth and from $n = 8$ onward it is faster to perform the static checks.

Fig. 11 shows the timings for Synthetic Experiment 2: they are nearly constant at, on average, 0.7 s. In both, Synthetic Experiment 1 and 2, the time for applicability checking is nearly constant. We refrain from increasing n further in Synthetic Experiment 1, because overall flattening becomes infeasible.

Our synthetic experiments show that applicability checking scales compared to either flattening or overall flattening. In fact, the timings are near constant for our experiments. We conjecture that (i) SAT solving is efficient enough to handle formulas with the resulting number of variables and that (ii) the formulas have a structure that is easy to solve.

11.6. Threats to validity

External validity. A threat to generalization of our results is that all code used in the case studies has been re-modeled by the authors themselves and consists of only 840 LoC. It should be noted that in two of the three case studies, the non-variable context was not considered (for example, in `FormbaR` less than 10% of the code was re-modeled). On the other hand, any reasonable product line is bound to have a substantial degree of commonality and there is no point to look at the invariant aspects of an application for the present evaluation.

Another threat is that, while VMs are a general concept for product lines, we only investigated its use for DOP, and only for the ABS language. We consider this to be unproblematic, because (i) DOP is a fairly general SPL approach that encompasses FOP (Schaefer and Damiani, 2010) and (ii) the ABS implementation of DOP follows general DOP principles (Schaefer and Damiani, 2010) and closely resembles other implementations, such as DOP in Java (Koscielny et al., 2014).

Internal validity. A threat concerning our conclusions is that, due to our aim of reusing code from the existing flattening and type checker implementation, we rely on pre-existing code and design choices for both flattening and applicability: we reuse the `Choco` solver part of the ABS compiler and do not optimize the flattening mechanisms.

Another threat is that the applicability checks take constant time due to the design of our examples. We mitigated this by using random feature models to exclude artifacts stemming from the structure of the feature model.

12. Related work

We focus on the approaches most relevant and closest to the VM concept. We refer to Apel et al. (2013) for a systematic introduction to feature-oriented SPLs, to Schaefer et al. (2012) for an overview of diverse system developments, to Thüm et al. (2014) for a classification and survey of analysis strategies for SPLs, and to Holl et al. (2012) for a systematic and expert survey on capabilities supporting MPLs.

12.1. Programming constructs for MPLs and variant interoperability

Schröter et al. (2013a) advocate the use of suitable interfaces to support compositional analysis of MPLs, consisting of FOP SPLs of Java programs, during different stages of the development process. Damiani et al. (2014b) 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 VM, the proposal does not support encapsulation at SPL level. More recently, Damiani et al. (2019) 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 (Igarashi et al., 1999, 2001). Their idea is to lift to the SPL level the use of dependent feature models to capture MPLs, as advocated by Schröter et al. (2016, 2013b). Like the earlier paper (Damiani et al., 2014b), the SPL construct proposed by Damiani et al. (2019) 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, the VM concept does not require feature model composition.

None of the proposals mentioned above support variant interoperability (Damiani et al., 2018b). Setyautami et al. (2018) address variant interoperability at the level 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 (Damiani et al., 2018b), where we consider 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 intertwined. 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.

Table 2
Results for the experiments in the case studies.

Model	Family-based static checking			Overall flattening	Flattening
	Normal form + type uniformity	Applicability	Pre-typing		
FormbaR-3	<0.01	<0.01	0.2	0.03	0.02
WMMF-3	<0.01	<0.01	0.9	1.1	0.05
AISCO-3	<0.01	<0.01	0.3	0.08	0.03

Table 3
Results for synthetic Experiment 1. Comparison of family-based static checking with overall flattening (times in seconds). Relative change is computed as $\frac{\text{total time for static checking}}{\text{time for overall flattening}}$.

# of features	Family-based static checking			Overall flattening	Relative change
	Normal form + type uniformity	Applicability	Pre-typing		
7	<0.01	0.05	0.07	0.07	41%
8	<0.01	0.05	0.08	0.20	−54%
9	<0.01	0.05	0.10	0.49	−227%
10	<0.01	0.06	0.08	1.63	−1 064%
11	<0.01	0.09	0.09	5.44	−2 922%
12	<0.01	0.08	0.11	27.27	−14 252%

12.2. Family-based checking for SPLs of java-like programs

According to Thüm et al. (2014), the approaches to the analysis of SPLs can be classified into three main categories: *product-based analyses*, which operate only on generated variants (or models of variants); *family-based analyses*, which operate only on the artifact base by exploiting the feature model and configuration knowledge to obtain results about all variants; and *feature-based analyses*, which operate on the building blocks of the different variants (feature modules in FOP and deltas in DOP) in isolation (without using the feature model and configuration knowledge) to derive results on all variants. We refer to Thüm et al. (2014) for a survey on SPL type checking. Here we review type checking approaches for FOP and DOP that are close to our proposal.

Thaker et al. (2007) informally illustrate the implementation of a family-based type checking for the AHEAD system (Batory et al., 2004). It comprises: (i) A family-feature-based step that computes for each class a stub (according to the terminology used in Section 9.1, these stubs represent a type uniform signature for the SPL) and compiles each feature module in the context of all stubs (thus performing checks corresponding to type uniformity and partial typing in our terminology) and (ii) a family-based step that infers a set of constraints that are combined with the feature model to generate a formula (modeling applicability and dependency) whose satisfiability should imply that all variants can be generated and successfully compile.

Delaware et al. (2009) formalize feature-family-based type checking for the Lightweight Feature Java (LFJ) calculus, which models FOP for the Lightweight Java (LJ) calculus by Strniša et al. (2007). It comprises: (i) A feature-based step that uses a constraint-based type system for LFJ to analyze each feature module in isolation and infer a set of constraints for each feature module and (ii) a family-based step where the feature model and the inferred constraints are used to generate a formula whose satisfiability implies that all variants can be generated and type check.

Bettini et al. (2013a) proposed a type checking approach for DOP. It comprises: (i) A feature-based analysis that uses a constraint-based type system for IFJ (a core calculus formalizing DOP for SPLs of Featherweight Java programs) to infer a type abstraction for each delta and (ii) a product-based step that uses these type abstractions to generate, for each product of the SPL, a type abstraction (of the associated variant) that is checked to establish whether the associated variant type checks. This approach has been enhanced by introducing a family-based step that builds a product family generation tree which is then traversed in order to perform optimized generation and check of type abstractions of all variants (Damiani and Schaefer, 2012), and has been partially implemented in a branch of the ABS tool chain (Damiani et al., 2017b).

Damiani and Lienhardt (2016) formalize, by means of IFJ, a feature-family-based type checking approach for DOP inspired by the approach for FOP by Thaker et al. (2007) and Delaware et al. (2009). It enforces type uniformity and comprises partial typing, applicability analysis and dependency analysis. Being feature-family-based, it represents an improvement over the previous type checking approaches for DOP, because it does not require to iterate over the set of all products. The family-based checks presented in this paper extend to ABS-VM the corresponding family-based type checks for IFJ (Damiani and Lienhardt, 2016).

12.3. Variability-aware module systems

Kästner et al. (2012b) propose a *variability-aware module system* (called VAMS in the following). Like in our proposal, each VAMS module is an SPL, however, VAMS and the VM concept differ fundamentally:

- VAMS is defined on top of the procedural system programming language C and the annotative approach (see Section 3) to SPL implementation (C, through its preprocessor directives `#define` and `#ifdef`, has built-in support for annotative SPLs).
- VMs are defined on top of the object-oriented modeling language ABS and the transformational (specifically, delta-oriented) approach to SPL implementation (ABS has built-in support for delta-oriented SPLs).

Due to this difference, also the design of VAMS and VM programs differs fundamentally. Therefore, it is not meaningful to compare both approaches directly in terms of examples and concrete case studies. Instead, we present a detailed comparison, where we focus on the four most important *conceptual* differences between the two approaches.

(i) VAMS does not encapsulate variability (Section 6): Modules import function declarations without specifying the modules from which they should be imported. To generate a variant VAMS requires the user to write a composition expression, which lists all modules to be composed and resolves dependencies and ambiguities (such as 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). Hence,

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, the VM concept achieves simplicity: 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. (2012b) do not mention this issue). Making two variants of the same module co-exist, requires to create a copy of the module and to rename (possibly by using the module composition language provided in VAMS) all its features and all its exported functions. In contrast, providing usable support to variant interoperability is a central design goal of VM.

(iii) Variability in VAMS 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 (Cardelli, 1997) 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 to procedural languages, where the interface of each module describes names and types of imported and exported functions, and there is a global function name space. Moreover, even though each module has its own feature model, there is a global feature name space. In contrast, the VM concept targets Java-like languages, it is based on the module system of ABS (Hähnle, 2013; Johnsen et al., 2010) (a fairly standard module system close to Java and Haskell), and implemented as an extension of the ABS module system. Each VM has a local name space (which reduces overhead), also features are local to VMs.

12.4. Variability modules in java

Setyautami and Hähnle (2021) suggest that the VM concept can be implemented on top of any Java-like language with modest effort. The solution 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 realized by 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 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, Setyautami and Hähnle (2021) does not feature a formal semantics of VM and family-based checking.

13. Conclusion and future work

This article introduced 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 co-exist and interoperate.

Central to the design of VM are simplicity and usability, specifically the PEV that makes dependencies on variants explicit, so that a simple Java- or Haskell-like module concept is sufficient.

We formalized the syntax and semantics of variability modules as an extension of the ABS language called ABS-VM. The semantics is given in terms of flattening rules that transform an ABS-VM program into a set of ABS modules. This made it possible to implement an ABS-VM compiler as a front-end to the ABS tool chain.

In addition to compilation, we defined and implemented PEV-compliance, type uniformity, pre-typing and applicability consistency as *family-based* checks for ABS-VM programs.

We evaluated the VM concept and our implementation of it quantitatively and qualitatively by case studies that were partly taken from industrial code used in production.

We are currently formalizing and implementing an extended analysis called *dependence consistency* checking: it aims to fully support family-based checking of type-safety (see the discussion at the beginning of Section 9). The extended tool chain will be subjected to validation by novel and even larger case studies.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Code linked from article.

Acknowledgments

We thank the anonymous SPLC 2021 and the JSS reviewers for insightful comments and suggestions that helped to improve this paper. This work was partially supported by the Research Council of Norway via SIRIUS (237898) and PeTWIN (294600).

Appendix A. Rules for delta application

We present the rules describing the application of an ordered sequence of deltas $\overline{\texttt{Dlt}}$ to a sequence of interface/class definitions $\overline{\texttt{Defn}}$. The rules, given in Fig. A.12, fall into the following categories:

- *Rules for a Sequence of Deltas* 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 extract the first interface/class operation from the delta and apply it to the list of definitions. D:End concludes the application process when the sequence of the deltas to be applied is empty.
- *Rules for a Delta* describe 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* modify the `extends` clauses of interfaces and `implements` clauses of classes by removing (D:EM:Rems) or adding (D:EM:Adds) an interface name. Rule D:EM:Empty is triggered when all modifications of the clause were applied.

Rules for a Sequence of Deltas

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

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

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

Rules for a Delta

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

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

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

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

Rules for Extends/Implements Clauses

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

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

Rules for Interfaces

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

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

Rules for Classes

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

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

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

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

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

Fig. A.12. Delta application rules.

- *Rules for Interfaces* 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 no header with this name is already present). Rule D:I:Rems removes an existing method header from the interface.
- *Rules for Classes* modify classes. They 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 method implementations (rules D:C:AddsM and D:C:RemsM). Second, methods may be modified using rule D:C:Mods. This rule replaces the method implementation, 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.

Variability-free ABS module typing

$$\text{T:Module}\quad \frac{\texttt{PrgS};\ \texttt{M} \vdash\ \overline{\texttt{Defn}} \qquad [\ \texttt{PrgS};\ \texttt{M};\ \emptyset \vdash\ \overline{\texttt{S}} : \Gamma\ ]}{\texttt{PrgS} \vdash \texttt{module M;}\ [\texttt{export tC;}]\ \ \texttt{import tC from M;}\ \overline{\texttt{Defn}}\ \ [\texttt{init}\ \{\ \overline{\texttt{S}}\ \}]}$$

Class definition typing

$$\text{T:Class}\quad \frac{\texttt{PrgS};\ \texttt{M};\ \{\ \texttt{this:M.C}\ \} \vdash\ \overline{\texttt{MD}} :\ \overline{\texttt{MBT}}}{\texttt{PrgS};\ \texttt{M} \vdash \texttt{class C}\ \ [\ \texttt{implements I}\ \ \overline{\texttt{I}}\ ]\ \{\ \overline{\texttt{FD}}\ \ \overline{\texttt{MD}}\ \}}$$

Method definition typing

$$\text{T:Meth}\quad \frac{\texttt{PrgS};\ \texttt{M};\ \{\ \texttt{this:M.C},\ \ \overline{\texttt{VBT x}}\ \} \vdash\ \overline{\texttt{S}} : \Gamma' \qquad \texttt{PrgS};\ \texttt{M};\ \Gamma' \vdash \texttt{E} : \texttt{EBT} \qquad \texttt{EBT} <: \texttt{VBT}}{\texttt{PrgS};\ \texttt{M};\ \{\ \texttt{this:M.C}\ \} \vdash \texttt{VBT m (}\ \overline{\texttt{VBT x}}\ \texttt{)}\ \{\ \overline{\texttt{S}}\ \texttt{return E}\} : (\ \overline{\texttt{VBT}}\ ) \to \texttt{VBT}}$$

Statement sequence typing

$$\text{T:Seq}\quad \frac{n \geq 0 \qquad \texttt{PrgS};\ \texttt{M};\ \Gamma_{i-1} \vdash \texttt{S}_i : \texttt{VBT}_i;\ \Gamma_i \quad (\text{for all } i \in \{1 \ldots n\})}{\texttt{PrgS};\ \texttt{M};\ \Gamma_0 \vdash \texttt{S}_1\texttt{;}\ldots\texttt{S}_n\texttt{;} : \Gamma_n}$$

Statement typing

$$\text{T:LocVarDec}\quad \frac{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{E} : \texttt{EBT} \qquad \texttt{EBT} <: \texttt{VBT}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{VBT x = E} : \texttt{VBT};\ \Gamma \uplus \{\texttt{x:VBT}\}}$$

$$\text{T:LocVarAssign}\quad \frac{\Gamma(\texttt{x}) = \texttt{VBT} \qquad \texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{E} : \texttt{EBT} \qquad \texttt{EBT} <: \texttt{VBT}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{x = E} : \texttt{VBT};\ \Gamma}$$

$$\text{T:FieldAssign}\quad \frac{\Gamma(\texttt{this})\texttt{=M.C} \qquad \texttt{PrgS(M)(C)(f)} = \texttt{VBT} \qquad \texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{E} : \texttt{EBT} \qquad \texttt{EBT} <: \texttt{VBT}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{this.f= E} : \texttt{VBT};\ \Gamma}$$

Expression typing

$$\text{T:Var}\quad \texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{x} : \Gamma(\texttt{x})$$

$$\text{T:Field}\quad \frac{\Gamma(\texttt{this})\texttt{=M.C} \qquad \texttt{PrgS(M)(C)(f)} = \texttt{VBT}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{this.f} : \texttt{VBT}}$$

$$\text{T:Invk}\quad \frac{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{E} : \texttt{M'.I} \qquad \texttt{aType(M'.I.m)} = (\ \overline{\texttt{VBT}}\ ) \to \texttt{VBT} \qquad \texttt{PrgS};\ \texttt{M};\ \Gamma \vdash\ \overline{\texttt{E}} :\ \overline{\texttt{VBT'}} \qquad \overline{\texttt{VBT'}} <:\ \overline{\texttt{VBT}}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{E.m(}\ \overline{\texttt{E}}\ \texttt{)} : \texttt{VBT}}$$

$$\text{T:New}\quad \frac{\texttt{C} \in \texttt{dom(PrgS(M'))}}{\texttt{PrgS};\ \texttt{M};\ \Gamma \vdash \texttt{new M'.C()} : \texttt{M'.C}}$$

Interface definition typing

$$\text{T:Interface}\quad \texttt{PrgS};\ \texttt{M} \vdash \texttt{ID}$$

Fig. B.13. Variability-free ABS module typing rules.

Appendix B. Rules for Vf-ABS module typing

We present the rules of the type system for variability-free ABS modules. The rules, given in Fig. B.13, use the following lookup function for retrieving the type of an attribute `M.N.a` in the program signature table `PrgS` (recall that an attribute can be either a method or a field):

$$\texttt{aType(M.N.a)} = \begin{cases} \texttt{PrgS(M)(N)(a)} & \text{if } \texttt{a} \in \texttt{dom(PrgS(M)(N))} \\ \texttt{aType(M'.I.a)} & \text{if } \texttt{a} \notin \texttt{dom(PrgS(M)(N))}, \\ & \texttt{M'.I} \in \texttt{PrgS(M)(N)(super)} \end{cases}$$

For sake of simplicity, in the rule premises we use typing a sequence of elements as a shorthand for the implicit list of typing each element (in T:Module, T:Class, T:Invk). The rules fall into the following categories.

- *Variability-free ABS module typing* Rule T:Module is fairly standard: each class/interface declaration in the module is typed relative to the signature of the program and the name of the module. The sequence of statements in the (optional) `init`-block is typed relative to the signature of the program, the name of the module and the empty typing environment (environment Γ is discarded). No checks on the `import`- and `export`-clauses are performed, because these are subsumed by the assumption that the signature `PrgS` is well-formed.
- *Class definition typing* Rule T:Class is fairly standard. No checks on the field declarations and on the `implements`-clause are performed, because these are subsumed by the assumption that the signature `PrgS` is well-formed.
- *Method definition typing* Rule T:Meth is fairly standard. First, it types the statements and the returned expression `e` in the

VM pre-typing

PT:VM

$$\frac{\begin{array}{ccc}\textbf{Condition1} & \textbf{Condition2} & \textbf{Condition3}\\ \texttt{FPrgS};\ \texttt{M} \Vdash \overline{\texttt{Defn}} & [\ \texttt{FPrgS};\ \texttt{M};\ \emptyset \Vdash \overline{\texttt{S}} : \Gamma\ ] & \texttt{FPrgS};\ \texttt{M} \Vdash \overline{\texttt{Dlt}}\end{array}}{\texttt{FPrgS} \Vdash \texttt{MdlH}\ \overline{\texttt{Defn}}\ [\texttt{init}\ \{\ \overline{\texttt{S}}\ \}]\ \overline{\texttt{Dlt}}\ \texttt{CK}}$$

Condition1:

Each class/interface being modified/removed in a delta in $\overline{\texttt{Dlt}}$ is declared in $\overline{\texttt{Defn}}$ or added by some delta in $\overline{\texttt{Dlt}}$.

Condition2:

Each attribute being modified in/removed from a class C by a delta in $\overline{\texttt{Dlt}}$ is declared in C in $\overline{\texttt{Defn}}$ or declared/added in C by some delta in $\overline{\texttt{Dlt}}$.

Condition3:

Each method signature being removed from an interface I by a delta in $\overline{\texttt{Dlt}}$ is declared in I in $\overline{\texttt{Defn}}$ or declared/added in I by some delta in $\overline{\texttt{Dlt}}$.

Fig. C.14. VM pre-typing rule.

body of the method with respect to the appropriate typing environments. Then it checks that the type of `e` is a subtype of return type of the method.

- *Statement sequence typing* Rule T:Seq is fairly standard: a sequence of $n \geq 0$ statements is typed relative to the signature of the program, the name of the module, and a typing environment Γ_0. The ith ($i \geq 1$) statement of the sequence is typed relative to Γ_i and produces Γ_{i+1}. The rule produces a typing environment Γ_n that is augmented by the type assumptions for the variables declared by the statements in the sequence.
- *Statement typing* These rules are fairly standard: a statement is typed relative to the signature of the program, the name of the module, a typing environment. Each rule produces a typing environment that is either augmented by the type assumption for the variable declared by the statement (if the statement is a local variable declaration) or unchanged (otherwise).
- *Expression typing* These rules are fairly standard: each expression is typed relative to the signature of the program, the name of the module, and a typing environment Γ that assigns a variable basic type to each variable that may occur in the expression and, whenever the expression occurs in the body of a method of a class `C`, assigns type `C` to `this`. We just point out that: fields can only be accessed through `this` (see rule T:Field), and methods cannot be invoked in `new`-expressions (rules T:New and T:Invk).
- *Interface definition typing* The assumption that the signature `PrgS` (of the program which contains the module `M` where the interface declaration `ID` occurs) is well-formed (Definition 18) entails that the interface declaration `ID` is well typed. Therefore, rule T:Interface performs no checks.

Appendix C. Rules for VM pre-typing

We present the pre-typing rules for VMs. The rules, given in Figs. C.14–C.16 use the following lookup function for retrieving the type of an attribute `M.N.a` in the family program signature table `FPrgS`:

$$\texttt{aType(M.N.a)} = \begin{cases} \texttt{FPrgS(M)(N)(a)} & \text{if } \texttt{a} \in \texttt{dom(FPrgS(M)(N))} \\ \texttt{aType(M'.I.a)} & \text{if } \texttt{a} \notin \texttt{dom(FPrgS(M)(N))} \text{ and} \\ & \texttt{M'.I [with ...]} \in \texttt{FPrgS(M)(N)(super)} \end{cases}$$

Rule PT:VM in Fig. C.14 first checks three consistency conditions, then it pre-types each interface/class definition in the module core part (Fig. C.15), the sequence of statements in the (optional) `init`-block, and each delta in the module delta part (Fig. C.16). No checks on configuration knowledge are performed, because these are subsumed by the assumption that the family program signature `FPrgS` is type uniform.

The rules in Fig. C.15 are a straightforward adaptation of the typing rules for variability-free ABS classes an interfaces (Fig. B.13) and fall into corresponding categories: *Interface definition pre-typing*, *Expression pre-typing* (Rule PT:Original is for pre-tying the `original` method invocation, which may occur in a method addition operation), *Statement pre-typing*, *Statement sequence pre-typing*, *Method definition pre-typing*, *Class definition pre-typing*.

The rules in Fig. C.16 fall into the following categories:

- *Delta pre-typing* Rule PT:Delta simply pre-types each class operation. No checks are performed on interface operations, because these are subsumed by the assumption that the family program signature `FPrgS` is type uniform.
- *Class operation pre-typing* Rule PT:Adds-Class simply pre-types the added class definition relying on rule PT:Class of Fig. C.15. Similar as before, rule PT:Removes-Class performs no checks, because these are subsumed by the assumption that the family program signature `FPrgS` is type uniform. Rule PT:Modifies-Class pre-types each attribute operation.
- *Attribute operation pre-typing* Rule PT:Adds-Meth simply pre-types the added method definition relying on rule PT:Meth of Fig. C.15. Rules PT:Adds-Field, PT:Removes-Attr perform no checks, because these are subsumed by the assumption that the family program signature `FPrgS` is type uniform. Rule PT:Modifies-Meth is similar to rule PT:Adds-Meth, however, it pre-types the body of the modified method relative to an environment that assigns to `original` the type of the method being modified. This makes it possible to use rule PT:Original of Fig. C.15 during pre-typing of $\overline{\texttt{S}}$.

Class definition pre-typing

$$\textsc{PT:Class}\ \frac{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \overline{\texttt{MD}} : \overline{\texttt{MT}}}{\texttt{FPrgS; M} \Vdash \texttt{class C [implements I } \overline{\texttt{I}} \texttt{] \{ } \overline{\texttt{FD}}\ \overline{\texttt{MD}} \texttt{ \}}}$$

Method definition pre-typing

$$\textsc{PT:Meth}\ \frac{\texttt{FPrgS; M; \{ this:M.C, } \overline{\texttt{VT x}} \texttt{ \}} \Vdash \overline{\texttt{S}} : \Gamma' \qquad \texttt{FPrgS; M; } \Gamma' \Vdash \texttt{E} : \texttt{ET} \qquad \mathsf{strip}(\texttt{ET}) <: \mathsf{strip}(\texttt{VT})}{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \texttt{VT m (} \overline{\texttt{VT x}} \texttt{) \{} \overline{\texttt{S}} \texttt{ return E\}} : (\overline{\texttt{VT}}) \to \texttt{VT}}$$

Statement sequence pre-typing

$$\textsc{PT:Seq}\ \frac{n \geq 0 \qquad \texttt{FPrgS; M;} \Gamma_{i-1} \Vdash \texttt{S}_i : \texttt{VT}_i; \Gamma_i \quad (\text{for all } i \in \{1 \ldots n\})}{\texttt{FPrgS; M;} \Gamma_0 \Vdash \texttt{S}_1\texttt{;}\ldots\texttt{S}_n\texttt{;} : \Gamma_n}$$

Statement pre-typing

$$\textsc{PT:LocVarDec}\ \frac{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{E} : \texttt{ET} \qquad \mathsf{strip}(\texttt{ET}) <: \mathsf{strip}(\texttt{VT})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{VT x = E} : \texttt{VT;} \Gamma \uplus \{\texttt{x:VT}\}}$$

$$\textsc{PT:LocVarAssign}\ \frac{\Gamma(\texttt{x}) = \texttt{VT} \qquad \texttt{FPrgS; M;} \Gamma \Vdash \texttt{E} : \texttt{ET} \qquad \mathsf{strip}(\texttt{ET}) <: \mathsf{strip}(\texttt{VT})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{x = E} : \texttt{VT;} \Gamma}$$

$$\textsc{PT:FieldAssign}\ \frac{\Gamma(\texttt{this})\texttt{=M.C} \qquad \texttt{FPrgS(M)(C)(f)} = \texttt{VT} \qquad \texttt{FPrgS; M;} \Gamma \Vdash \texttt{E} : \texttt{ET} \qquad \mathsf{strip}(\texttt{ET}) <: \mathsf{strip}(\texttt{VT})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{this.f= E} : \texttt{VT;} \Gamma}$$

Expression pre-typing

$$\textsc{PT:Var}\ \texttt{FPrgS; M;} \Gamma \Vdash \texttt{x} : \Gamma(\texttt{x}) \qquad \textsc{PT:Field}\ \frac{\Gamma(\texttt{this})\texttt{=M.C} \qquad \texttt{FPrgS(M)(C)(f)} = \texttt{VT}}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{this.f} : \texttt{VT}}$$

$$\textsc{PT:Invk}\ \frac{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{E} : \texttt{M'.I [with ...]} \qquad \mathsf{aType}(\texttt{M'.I.m}) = (\overline{\texttt{VT}}) \to \texttt{VT} \qquad \texttt{FPrgS; M;} \Gamma \Vdash \overline{\texttt{E}} : \overline{\texttt{VT'}} \qquad \mathsf{strip}(\overline{\texttt{VT'}}) <: \mathsf{strip}(\overline{\texttt{VT}})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{E.m(} \overline{\texttt{E}} \texttt{)} : \texttt{VT}}$$

$$\textsc{PT:New}\ \frac{\texttt{C} \in \mathsf{dom}(\texttt{FPrgS(M')})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{new M'.C() [with KE]} : \texttt{M'.C [with KE]}}$$

$$\textsc{PT:Original}\ \frac{\Gamma(\texttt{original}) = (\overline{\texttt{VT}}) \to \texttt{VT} \qquad \texttt{FPrgS; M;} \Gamma \Vdash \overline{\texttt{E}} : \overline{\texttt{VT'}} \qquad \mathsf{strip}(\overline{\texttt{VT'}}) <: \mathsf{strip}(\overline{\texttt{VT}})}{\texttt{FPrgS; M;} \Gamma \Vdash \texttt{original(} \overline{\texttt{E}} \texttt{)} : \texttt{VT}}$$

Interface definition pre-typing

$$\textsc{PT:Interface}\ \texttt{FPrgS; M} \Vdash \texttt{ID}$$

Fig. C.15. Class/interface pre-typing rules.

Delta pre-typing

$$\textsc{PT:Delta}\ \frac{\texttt{FPrgS; M} \Vdash \overline{\texttt{CO}}}{\texttt{FPrgS; M} \Vdash \texttt{delta D;}\ \overline{\texttt{CO}}\ \overline{\texttt{IO}}}$$

Class operation pre-typing

$$\textsc{PT:Adds-Class}\ \frac{\texttt{FPrgS; M} \Vdash \texttt{CD}}{\texttt{FPrgS; M} \Vdash \texttt{adds CD}} \qquad \textsc{PT:Removes-Class}\ \texttt{FPrgS; M} \Vdash \texttt{removes C}$$

$$\textsc{PT:Modifies-Class}\ \frac{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \overline{\texttt{AO}}}{\texttt{FPrgS; M} \Vdash \texttt{modifies class C [adds IR } \overline{\texttt{IR}} \texttt{] [removes IR } \overline{\texttt{IR}} \texttt{] \{ } \overline{\texttt{AO}} \texttt{ \}}}$$

Attribute operation pre-typing

$$\textsc{PT:Adds-Meth}\ \frac{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \texttt{MD} : \texttt{MT}}{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \texttt{adds MD}} \qquad \textsc{PT:Adds-Field}\ \texttt{FPrgS; M} \Vdash \texttt{adds FD} \qquad \textsc{PT:Removes-Attr}\ \texttt{FPrgS; M} \Vdash \texttt{removes AD}$$

$$\textsc{PT:Modifies-Meth}\ \frac{\texttt{FPrgS; M; \{ this:M.C, original:(} \overline{\texttt{VT}} \texttt{) } \to \texttt{VT, } \overline{\texttt{VT x}} \texttt{ \}} \Vdash \overline{\texttt{S}} : \Gamma' \qquad \texttt{FPrgS; M;} \Gamma' \Vdash \texttt{E} : \texttt{ET} \qquad \mathsf{strip}(\texttt{ET}) <: \mathsf{strip}(\texttt{VT})}{\texttt{FPrgS; M; \{ this:M.C \}} \Vdash \texttt{modifies VT m (} \overline{\texttt{VT x}} \texttt{) \{} \overline{\texttt{S}} \texttt{ return E\}}}$$

Fig. C.16. Delta pre-typing rules.

Appendix D. Rules for applicability constraint inference

We present the rules for computing the applicability constraint of an ABS program. The rules parse a normal form ABS-VM program `Prg` from left to right. They assume the list of deltas in `Prg` is in an order compatible with the configuration knowledge `CK`.

All statements in the focus of the rules have the form $C \vdash \mathtt{G} : \Theta, \Phi$, where (i) `G` is the syntax element being parsed according to Fig. 6, (ii) C stores relevant context information (the currently parsed module's name, the currently parsed delta's activation condition or **true** when parsing the module core part, the input declaration presence mapping), (iii) Θ is the output declaration presence mapping (from the analysis of `G`), (iv) Φ is the output applicability constraint (from the analysis of `G`). Elements Φ and Θ are defined simultaneously. At each stage Φ contains information about current applicability of deltas in products, while Θ collects information about the presence of `G` in products.

The first applicability rule is for programs:

$$\text{A:PRG}\quad \frac{\mathtt{Prg} = \mathtt{Mdl}_1\ \cdots\ \mathtt{Mdl}_n \qquad \vdash \mathtt{Mdl}_i : \Theta_i, \Phi_i}{\vdash \mathtt{Prg} : \bigcup_{1\le i\le n} \Theta_i,\ \bigwedge_{1\le i\le n} \Phi_i}$$

The rule runs through each module declaration $\mathtt{Mdl}_i$, collecting presence and applicability information. According to Fig. 6, each class/interface operation `CO`/`IO` declares the *unqualified* class/interface name `C`/`I` on which it intends to act. If the operation is in module `M` then, in accordance with the flattening rules, it is (implicitly) qualified by `M`. Therefore, deltas cannot declare operations outside of the scope of their module, so no module can modify presence conditions in other modules.

The next rule is for modules. The initial domain of the presence mapping Θ_0 is set to **false**. This domain includes the names of all interfaces, classes, and attributes extracted from the module by fsig.

$$\text{A:MODULE}\quad \frac{\Theta_0 = [\mathtt{Q} \mapsto \mathbf{false}]_{\mathtt{Q}\in\mathrm{ddom}(\mathtt{M})} \qquad \mathtt{M}, \Theta_{i-1}, \mathbf{true} \vdash \mathtt{Defn}_i : \Theta_i, \Phi_i \quad (1 \le i \le n) \qquad \mathtt{M}, \Theta_{i-1} \vdash \mathtt{Dlt}_i : \Theta_i, \Phi_i \quad (n+1 \le i \le m)}{\vdash \left(\begin{array}{l}\mathtt{module\ M;}\\ \mathtt{[\ export\ tC;\ ]}\quad \overline{\mathtt{import\ tC\ from\ M;}}\quad \mathtt{[\ features\ \overline{F}\ with\ \Phi;\ ]}\quad \overline{\mathtt{KD}}\ \overline{\mathtt{PD}}\\ \mathtt{Defn}_1\ \cdots\ \mathtt{Defn}_n\quad \mathtt{[\ init\ \{\ \overline{S}\}\ ]}\quad \mathtt{Dlt}_{n+1}\ \cdots\ \mathtt{Dlt}_m\quad \mathtt{CK}\end{array}\right) : \Theta_m,\ \bigwedge_{1\le i\le m} \Phi_i}$$

The judgments for class and interface declarations have an additional constraint **true** on the left-hand side: it allows to reuse the rules for deltas adding declarations where we set **true** as activation condition. The applicability constraint of the module is the conjunction of all Φ_i, while the presence mapping is updated sequentially.

The next two rules handle class/interface declarations uniformly, whether they come from the core part or from deltas (see rule [A:ADDC] below).

$$\text{A:CLASS}\quad \frac{\mathtt{M.C}, \Theta_{i-1}, \Phi \vdash \mathtt{FD}_i : \Theta_i, \Phi_i \quad (1 \le i \le n) \qquad \mathtt{M.C}, \Theta_{i-1}, \Phi \vdash \mathtt{MD}_i : \Theta_i, \Phi_i \quad (n+1 \le i \le m) \qquad \Theta' = \Theta_m[\mathtt{M.C} \mapsto \Theta_m(\mathtt{M.C}) \vee \Phi] \qquad \Phi' = (\textstyle\bigwedge_{1\le i\le m} \Phi_i) \wedge (\Phi \to \neg\Theta_m(\mathtt{M.C}))}{\mathtt{M}, \Theta_0, \Phi \vdash \mathtt{[\ unique\ ]}\ \ \mathtt{class\ C\ implements\ IR}_1 \cdots \mathtt{IR}_r\ \{\ \mathtt{FD}_1 \cdots \mathtt{FD}_n\ \mathtt{MD}_{n+1} \cdots \mathtt{MD}_m\} : \Theta', \Phi'}$$

$$\text{A:INTERFACE}\quad \frac{\mathtt{M.I}, \Theta_{i-1}, \Phi \vdash \mathtt{MH}_i : \Theta_i, \Phi_i \quad (1 \le i \le m) \qquad \Theta' = \Theta_m[\mathtt{M.I} \mapsto \Theta_m(\mathtt{M.I}) \vee \Phi] \qquad \Phi' = (\textstyle\bigwedge_{1\le i\le m} \Phi_i) \wedge (\Phi \to \neg\Theta_m(\mathtt{M.I}))}{\mathtt{M}, \Theta_0, \Phi \vdash \mathtt{[\ unique\ ]}\ \ \mathtt{interface\ I\ extends\ IR}_1 \cdots \mathtt{IR}_r\ \{\ \mathtt{MH}_1 \cdots, \mathtt{MH}_m\} : \Theta', \Phi'}$$

We do not collect information about the presence of implements/extends, because these clauses can be always added/removed, so they do not influence delta applicability and flattening.

The rules for fields, methods, and method headers collect the expected information.

$$\text{A:FIELD}\quad \mathtt{M.C}, \Theta, \Phi \vdash \mathtt{T\ x;} : \Theta[\mathtt{M.C.x} \mapsto \Theta(\mathtt{M.C.x}) \vee \Phi], \Phi \Rightarrow (\neg\Theta(\mathtt{M.C.x}))$$

$$\text{A:METH}\quad \frac{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{MH} : \Theta', \Phi'}{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{MH\ \{\ \overline{S}\ return\ E;\ \}} : \Theta', \Phi'}$$

$$\text{A:MHEADER}\quad \mathtt{M.N}, \Theta, \Phi \vdash \mathtt{T\ m(\ \overline{T\ x}\ )} : \Theta[\mathtt{M.N.m} \mapsto \Theta(\mathtt{M.N.m}) \vee \Phi], \Phi \Rightarrow (\neg\Theta(\mathtt{M.N.m}))$$

The next rule parses deltas. It traverses the class/interface operations, using information about the application condition `M.D` on the left-hand side.

$$\text{A:DELTA}\quad \frac{\mathtt{M}, \Theta_{i-1}, \mathtt{M.D} \vdash \mathtt{CO}_i : \Theta_i, \Phi_i \quad (1 \le i \le n) \qquad \mathtt{M}, \Theta_{i-1}, \mathtt{M.D} \vdash \mathtt{IO}_i : \Theta_i, \Phi_i \quad (n+1 \le i \le m)}{\mathtt{M}, \Theta_0 \vdash \mathtt{delta\ M.D;\ CO}_1 \cdots \mathtt{CO}_n\ \mathtt{IO}_{n+1} \cdots \mathtt{IO}_m : \Theta_m, \textstyle\bigwedge_{1\le i\le m} \Phi_i}$$

The rule for adding a class simply forwards to the class declaration rule:

$$\text{A:AddC}\quad \frac{\mathtt{M}, \Theta, \Phi \vdash \mathtt{CD} : \Theta', \Phi}{\mathtt{M}, \Theta, \Phi \vdash \mathtt{adds\ CD} : \Theta', \Phi}$$

The rule for removing a class first states, in the constraint Φ', that for this operation to be valid the class `M.C` must be already be declared. It then updates the mapping Θ into Θ', stating that `M.C`, its fields, methods and subtype declarations are only present when that remove operation (triggered by the constraint in Φ) is not activated.

$$\text{A:RemC}\quad \frac{\Phi' = (\Phi \Rightarrow \Theta(\mathtt{M.C}))\quad \Theta' = \Theta[\mathtt{M.C} \mapsto \Theta(\mathtt{M.C}) \wedge (\neg\Phi)][\mathtt{M.C.a} \mapsto \Theta(\mathtt{M.C.a}) \wedge (\neg\Phi)]_{\mathtt{M.C.a} \in \mathsf{dom}(\Theta)}}{\mathtt{M}, \Theta, \Phi \vdash \mathtt{removes\ class\ C} : \Theta', \Phi'}$$

The rule for class modification ignores added/removed interfaces to implement, because this is always allowed. It generates the applicability constraint Φ' by accumulating the constraints Φ_i (all added elements must not be present before and all removed element must be present), and adding the constraint that the class `M.C` must be present.

$$\text{A:ModC}\quad \frac{\mathtt{M.C}, \Theta_{i-1}, \Phi \vdash \mathtt{AO}_i : \Theta_i, \Phi_i \quad (1 \le i \le n) \qquad \Phi' = (\bigwedge_{1 \le i \le n} \Phi_i) \wedge (\Phi \Rightarrow \Theta(\mathtt{M.C}))}{\mathtt{M}, \Theta_0, \Phi \vdash \mathtt{modifies\ class\ C\ adds\ IR}_1 \cdots \mathtt{IR}_q\ \mathtt{removes\ IR}'_1 \ldots \mathtt{IR}'_r\ \{\ \mathtt{AO}_1 \cdots \mathtt{AO}_n\ \} : \Theta_n, \Phi'}$$

The rule for adding attributes forwards to the rule for attribute definition:

$$\text{A:AddA}\quad \frac{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{AD} : \Theta', \Phi'}{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{adds\ AD} : \Theta', \Phi'}$$

The two following rules, for removing a field or a method, build a constraint Φ' ensuring that, whenever the activation condition Φ of the delta containing the operation becomes **true**, then the attribute is present ($\Theta(\mathtt{M.C.x})$ holds). Moreover, they update Θ' to store the information that the attribute is no longer available after the operation is applied.

$$\text{A:RemFD}\quad \frac{\Phi' = (\Phi \Rightarrow \Theta(\mathtt{M.N.x}))\quad \Theta' = \Theta[\mathtt{M.N.x} \mapsto \Theta(\mathtt{M.N.x}) \wedge (\neg\Phi)]}{\mathtt{M.N}, \Theta, \Phi \vdash \mathtt{removes\ VT\ x;} : \Theta', \Phi'}$$

$$\text{A:RemMH}\quad \frac{\Phi' = (\Phi \Rightarrow \Theta(\mathtt{M.C.m}))\quad \Theta' = \Theta[\mathtt{M.C.m} \mapsto \Theta(\mathtt{M.C.m}) \wedge (\neg\Phi)]}{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{removes\ VT\ m(}\ \overline{\mathtt{VT\ x}}\ \mathtt{)} : \Theta', \Phi'}$$

The rule for modifying a method requires in the applicability constraint Φ' that the method must already be present (when activation condition Φ holds), but leaves the presence mapping unchanged:

$$\text{A:ModM}\quad \frac{\Phi' = (\Phi \Rightarrow \Theta(\mathtt{M.C.m}))}{\mathtt{M.C}, \Theta, \Phi \vdash \mathtt{modifies\ VT\ m(}\ \overline{\mathtt{VT\ x}}\ \mathtt{)}\ \{\ \overline{\mathtt{S}}\ \mathtt{return\ E;}\ \} : \Theta, \Phi'}$$

The rules for delta operations on interfaces are similar to those for class operations:

$$\text{A:AddI}\quad \frac{\mathtt{M}, \Theta, \Phi \vdash \mathtt{ID} : \Theta', \Phi}{\mathtt{M}, \Theta, \Phi \vdash \mathtt{adds\ ID} : \Theta', \Phi}$$

$$\text{A:RemI}\quad \frac{\Phi' = (\Phi \Rightarrow \Theta(\mathtt{M.I}))\quad \Theta' = \Theta[\mathtt{M.I} \mapsto \Theta(\mathtt{M.I}) \wedge (\neg\Phi)][\mathtt{M.I.a} \mapsto \Theta(\mathtt{M.I.a}) \wedge (\neg\Phi)]_{\mathtt{M.I.a} \in \mathsf{dom}(\Theta)}}{\mathtt{M}, \Theta, \Phi \vdash \mathtt{removes\ interface\ I} : \Theta', \Phi'}$$

$$\text{A:ModI}\quad \frac{\mathtt{M.I}, \Theta_{i-1}, \Phi \vdash \mathtt{HO}_i : \Theta_i, \Phi_i \qquad \Phi' = (\bigwedge_{1 \le i \le n} \Phi_i) \wedge (\Phi \Rightarrow \Theta(\mathtt{M.I}))}{\mathtt{M}, \Theta_0, \Phi \vdash \mathtt{modifies\ interface\ I\ adds\ IR}_1 \ldots \mathtt{IR}_q\ \mathtt{removes\ IR}'_1 \cdots \mathtt{IR}'_r\ \{\ \mathtt{HO}_1 \cdots \mathtt{HO}_n\ \} : \Theta_n, \Phi'}$$

$$\text{A:AddMH}\quad \frac{\mathtt{M.I}, \Theta, \Phi \vdash \mathtt{MH} : \Theta', \Phi'}{\mathtt{M.I}, \Theta, \Phi \vdash \mathtt{adds\ MH} : \Theta', \Phi'}$$

Appendix E. Proof of Theorem 2

The following lemmas analyze the applicability consistency rules. Rules are grouped according to the properties they ensure.

E.1. Attributes

Lemma 3 (*Adding Attributes to Classes/interfaces*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ *have domain* ddom(M)*. Let* $M.N$ *be a class/interface such that* $M.N, \Theta, \Phi \vdash G : \Theta', \Phi'$ *holds, where* $G \in \{$FD, MD, MH, adds AD, adds MH$\}$ *(see Fig. 6). Let* a *be the attribute* G *is adding to* $M.N$ *and let* π *be a product of* M*. Then, the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *implies* *eval*$(\pi, \Theta(M.N.a), \text{Prg}) =$ **false**.
2. *eval*$(\pi, \Theta'(M.N.a), \text{Prg}) =$ **true** *if and only if* *either* *eval*$(\pi, \Theta(M.N.a), \text{Prg}) =$ **true** *or* *eval*$(\pi, \Phi, \text{Prg}) =$ **true**. *Moreover,* $\Theta'(x) = \Theta(x)$*, for all* $x \in$ ddom(M) *such that* $x \neq M.N.a$.

Proof. In these rules Φ is an activation condition (**true** for adding to the core), while Φ' is an applicability constraint. The proof is by verifying the statements for rules [A:Field], [A:Meth], [A:MHeader], [A:AddA], [A:AddMH]. □

Lemma 4 (*Removing Attributes from Classes/interfaces*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ *have domain* ddom(M)*. Let* $M.N$ *be a class/interface such that* $M.N, \Theta, \Phi \vdash G : \Theta', \Phi'$ *holds, where* $G \in \{$removes HD, removes MH$\}$ *(see Fig. 6). Let* a *be the attribute that* G *removes from* $M.N$ *and let* π *be a product of* M*. Then, the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *implies* *eval*$(\pi, \Theta(M.N.a), \text{Prg}) =$ **true**.
2. *eval*$(\pi, \Theta'(M.N.a), \text{Prg}) =$ **true** *if and only if* *both* *eval*$(\pi, \Theta(M.N.a), \text{Prg}) =$ **true** *and* *eval*$(\pi, \Phi, \text{Prg}) =$ **false**. *Moreover,* $\Theta'(x) = \Theta(x)$ *for all* $x \in$ ddom(M) *such that* $x \neq M.N.a$.

Proof. As before Φ is an activation condition (removing is never done in the core), while Φ' is an applicability constraint. The proof is by verifying the statements for the rules [A:RemFD] and [A:RemMH]. □

Lemma 5 (*Modifying Methods in Classes*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ *have domain* ddom(M)*. Let* $M.C$ *be a class such that* $M.C, \Theta, \Phi \vdash$ modifies $VT\ m(\overline{VT\ x})\ \{\ \overline{S}$ return E; $\} : \Theta', \Phi'$ *holds. If* π *is a product of* M *then the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *implies* *eval*$(\pi, \Theta(M.N.m), \text{Prg}) =$ **true**.
2. $\Theta' = \Theta$.

Proof. Here Φ is the activation condition of a delta. The proof is by verifying the statements for rule [A:ModM]. □

E.2. Classes/interfaces

Lemma 6 (*Adding Classes/interfaces to Modules*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ *have domain* ddom(M)*. Assume* $M, \Theta, \Phi \vdash G : \Theta', \Phi'$ *holds, where* $G \in \{$CD, ID, adds CD, adds ID$\}$ *(see Fig. 6). Let* N *be the class/interface that* G *is adding to* M *and let* π *be a product of* M*.*

Consider $\{Z_1, \ldots, Z_m\}$*, where* $Z_i \in \{$FD, MD, MH$\}$ *is the body of* N*, such that in the derivation of* G *above there are sub-derivations of* $M.I, \Theta_{i-1}, \Phi \vdash Z_i : \Theta_i, \Phi_i$ *for* $1 \leq i \leq m$*. Then, the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \bigwedge_{1 \leq i \leq m} \Phi_i, \text{Prg}) =$ **true** *and* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *both imply that* *eval*$(\pi, \Theta(M.N), \text{Prg}) =$ **false**.
2. *eval*$(\pi, \Theta'(M.N), \text{Prg}) =$ **true** *if and only if* *either* *eval*$(\pi, \Theta(M.N), \text{Prg}) =$ **true** *or* *eval*$(\pi, \Phi, \text{Prg}) =$ **true**. *Moreover,* $\Theta'(x) = \Theta_m(x)$ *for all* $x \in$ ddom(M) *such that* $x \neq M.N$.

Proof. As before, Φ is an activation condition (and **true** in the core). The proof uses Lemma 3 on the derivation ending by rule applications [A:Class], [A:Interface], [A:AddC], [A:AddI]. □

Lemma 7 (*Removing Classes/interfaces from Modules*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ *have domain* ddom(M)*. Assume* $M, \Theta, \Phi \vdash G : \Theta', \Phi'$ *holds, where* $G \in \{$removes class N, removes interface $N\}$*. Let* N *be the class/interface that* G *removes from* M *and let* π *be a product of* M*. Then, the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *implies* *eval*$(\pi, \Theta(M.N), \text{Prg}) =$ **true**.
2. *eval*$(\pi, \Theta'(M.N), \text{Prg}) =$ **true** *if and only if* *both* *eval*$(\pi, \Theta(M.N), \text{Prg}) =$ **true** *and* *eval*$(\pi, \Phi, \text{Prg}) =$ **false**.
3. *For all* $M.N.q \in \text{dom}(\Theta)$*:* *eval*$(\pi, \Theta'(M.N.q), \text{Prg}) =$ **true** *if and only if* *both* *eval*$(\pi, \Theta(M.N.q), \text{Prg}) =$ **true** *and* *eval*$(\pi, \Phi, \text{Prg}) =$ **false**. *Moreover, for all* $x \in$ ddom(M) *with* $x \notin M.N \cup \{M.N.q \mid M.N.q \in \text{dom}(\Theta)\}$ *we have* $\Theta'(x) = \Theta_m(x)$.

Proof. As before, Φ is an activation condition (no removal in the core). The proof is by verifying the statements for rules [A:RemC] and [A:RemI]. □

Lemma 8 (*Modifying Classes/interfaces in Modules*)**.** *Let* Prg *be a normal form ABS-VM program,* M *a module of* Prg*, and let* Θ *have domain* ddom(M)*. Assume*

$$M, \Theta, \Phi \vdash \text{modifies}\ \begin{matrix}\text{class } N \\ \text{interface } N\end{matrix}\ [\ \text{adds} \ldots\ ][\ \text{removes} \ldots\ ] \times \{Op_1 \cdots Op_n\} : \Theta', \Phi'$$

holds, where $Op_i \in \{$AO, SO$\}$*. In the premise of this derivation are sub-derivations of* $M.I, \Theta_{i-1}, \Phi \vdash Op_i : \Theta_i, \Phi_i$ *for* $1 \leq i \leq m$*. If* π *is a product of* M *then the following holds:*

1. *eval*$(\pi, \Phi', \text{Prg}) =$ **true** *if and only if* *eval*$(\pi, \bigwedge_{1 \leq i \leq n} \Phi_i, \text{Prg}) =$ **true** *and* *eval*$(\pi, \Phi, \text{Prg}) =$ **true** *both imply that* *eval*$(\pi, \Theta(M.N), \text{Prg}) =$ **true**.
2. $\Theta' = \Theta_n$.

Proof. Here Φ is the activation condition of a delta. The proof uses Lemmas 3–5 on the derivation ending in applications of rules [A:ModC] and [A:ModI]. □

E.3. Deltas and modules

Lemma 9 (*Applying Deltas, Building Modules*)**.** *Let* Prg *be a normal form ABS-VM program, let* M *be a module of* Prg*, and let* Θ_0 *have domain* ddom(M)*. Assume one of the following cases applies:*

- $M, \Theta_0 \vdash$ delta $M.D$; $\text{CO}_1 \cdots \text{CO}_n\ \text{IO}_{n+1} \cdots \text{IO}_m : \Theta', \Phi'$ *holds. Therefore, in the premises of this derivation we have* $M, \Theta_{i-1}, M.D \vdash \text{CO}_i : \Theta_i, \Phi_i$ *for* $1 \leq i \leq n$ *and* $M, \Theta_{i-1}, M.D \vdash \text{IO}_i : \Theta_i, \Phi_i$ *for* $n+1 \leq i \leq m$.

- $\vdash$ `module` M; $\cdots$ $\texttt{Defn}_1 \cdots \texttt{Defn}_n$ [`init`{ $\overline{\texttt{S}}$ }] $\texttt{Dlt}_{n+1} \cdots \texttt{Dlt}_m$ `CK` : Θ', Φ' *holds. Therefore, in the premises of this derivation we find:*
  1. $\Theta_0 = [Q \mapsto \textbf{false}]_{Q \in \mathrm{ddom}(M)}$
  2. $M, \Theta_{i-1}, \textbf{true} \vdash \texttt{Defn}_i : \Theta_i, \Phi_i$ *for* $1 \leq i \leq n$
  3. $M, \Theta_{i-1} \vdash \texttt{Dlt}_i : \Theta_i, \Phi_i$ *for* $n+1 \leq i \leq m$

If π *is a product of* M *then the following holds:*

1. $eval(\pi, \Phi', \texttt{Prg}) = \textbf{true}$ *if and only if* $eval(\pi, \Phi_i, \texttt{Prg}) = \textbf{true}$ *for all* i *such that* $1 \leq i \leq m$.
2. $\Theta' = \Theta_m$.

Proof. The proof of the first case uses Lemmas 6–8 on the derivation ending with application of rule [A:Delta]. The proof of the second case uses the first case and Lemma 6 on the derivation ending with application of rule [A:Module]. □

References

Alouneh, S., Abed, S., Shayeji, M.H.A., Mesleh, R., 2019. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. Artif. Intell. Rev. 52, 2575–2601.

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

Batory, D., 2005. Feature models, grammars, and propositional formulas. In: Proceedings of International Software Product Line Conference. SPLC, In: LNCS, vol. 3714, Springer, pp. 7–20. http://dx.doi.org/10.1007/11554844_3.

Batory, D., Sarvela, J.N., Rauschmayer, A., 2004. Scaling step-wise refinement. IEEE Trans. Softw. Eng. 30, 355–371. http://dx.doi.org/10.1109/TSE.2004.23.

Bettini, L., Damiani, F., 2017. Xtraitj: Traits for the Java platform. J. Syst. Softw. 131, 419–441. http://dx.doi.org/10.1016/j.jss.2016.07.035.

Bettini, L., Damiani, F., Schaefer, I., 2013a. Compositional type checking of delta-oriented software product lines. Acta Inform. 50 (2), 77–122. http://dx.doi.org/10.1007/s00236-012-0173-z.

Bettini, L., Damiani, F., Schaefer, I., Strocco, F., 2013b. TraitRecordJ: A programming language with traits and records. Sci. Comput. Program. 78 (5), 521–541. http://dx.doi.org/10.1016/j.scico.2011.06.007.

Bono, V., Damiani, F., Giachino, E., 2008. On traits and types in a java-like setting. In: Fifth IFIP International Conference on Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, September 7-10, 2008, Milano, Italy. pp. 367–382. http://dx.doi.org/10.1007/978-0-387-09680-3_25.

Bracha, G., 1992. The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance (Ph.D. thesis). Department of Comp. Sci., Univ. of Utah.

Cardelli, L., 1997. Program fragments, linking, and modularization. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '97, ACM, New York, NY, USA, pp. 266–277. http://dx.doi.org/10.1145/263699.263735, URL: http://doi.acm.org/10.1145/263699.263735.

Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R., 2010. Variability modelling in the ABS language. In: FMCO. In: LNCS, vol. 6957, Springer, pp. 204–224. http://dx.doi.org/10.1007/978-3-642-25271-6_11.

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

Czarnecki, K., Eisenecker, U.W., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., 2017a. A unified and formal programming model for deltas and traits. In: FASE. In: LNCS, vol. 10202, Springer, pp. 424–441. http://dx.doi.org/10.1007/978-3-662-54494-5_25.

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., 2018a. Interoperability of software product line variants. In: SPLC. ACM, pp. 264–268. http://dx.doi.org/10.1145/3233027.3236401.

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., 2018b. Same same but different: Interoperability of software product line variants. In: Principled Software Development. Springer, pp. 99–117. http://dx.doi.org/10.1007/978-3-319-98047-8_7.

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., Paolini, L., 2021. Variability modules for Java-like languages. In: Mousavi, M., Schobbens, P. (Eds.), Proc. 25th ACM Intl. Systems and Software Product Line Conf., Volume A. ACM, New York, NY, USA, pp. 1—12. http://dx.doi.org/10.1145/3461001.3471143.

Damiani, F., Lienhardt, M., 2016. On type checking delta-oriented product lines. In: Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings. In: LNCS, vol. 9681, Springer, pp. 47–62. http://dx.doi.org/10.1007/978-3-319-33693-0_4.

Damiani, F., Lienhardt, M., Muschevici, R., Schaefer, I., 2017b. An extension of the ABS toolchain with a mechanism for type checking SPLs. In: Integrated Formal Methods, 13th Intl. Conf., IFM, Turin, Italy. In: LNCS, vol. 10510, Springer, pp. 111–126. http://dx.doi.org/10.1007/978-3-319-66845-1_8.

Damiani, F., Lienhardt, M., Paolini, L., 2019. A formal model for multi software product lines. Sci. Comput. Program. 172, 203–231. http://dx.doi.org/10.1016/j.scico.2018.11.005.

Damiani, F., Schaefer, I., 2012. Family-based analysis of type safety for delta-oriented software product lines. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. In: LNCS, vol. 7609, Springer Berlin Heidelberg, pp. 193–207. http://dx.doi.org/10.1007/978-3-642-34026-0_15.

Damiani, F., Schaefer, I., Schuster, S., Winkelmann, T., 2014a. Delta-trait programming of software product lines. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 289–303. http://dx.doi.org/10.1007/978-3-662-45234-9_21.

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

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

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P., 2006. Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28 (2), 331–388. http://dx.doi.org/10.1145/1119479.1119483.

Hähnle, R., 2013. The abstract behavioral specification language: A tutorial introduction. In: Bonsangue, M., de Boer, F., Giachino, E., Hähnle, R. (Eds.), Intl. School on Formal Models for Components and Objects: Post Proceedings. In: LNCS, vol. 7866, Springer, pp. 1–37. http://dx.doi.org/10.1007/978-3-642-40615-7_1.

Holl, G., Grünbacher, P., Rabiser, R., 2012. A systematic review and an expert survey on capabilities supporting multi product lines. Inf. Softw. Technol. 54 (8), 828–852. http://dx.doi.org/10.1016/j.infsof.2012.02.002, URL: http://www.sciencedirect.com/science/article/pii/S095058491200033X.

Igarashi, A., Pierce, B.C., Wadler, P., 1999. Featherweight Java: A minimal core calculus for Java and GJ. In: Hailpern, B., Northrop, L.M., Berman, A.M. (Eds.), Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), Denver, Colorado, USA. ACM, pp. 132–146. http://dx.doi.org/10.1145/320384.320395.

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

Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M., 2010. 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. pp. 142–164. http://dx.doi.org/10.1007/978-3-642-25271-6_8.

Kamburjan, E., Hähnle, R., 2016. Uniform modeling of railway operations. In: FTSCS. In: Communications in Computer and Information Science, vol. 694, pp. 55–71. http://dx.doi.org/10.1007/978-3-319-53946-1_4.

Kamburjan, E., Hähnle, R., 2018. Prototyping formal system models with active objects. In: Bartoletti, M., Knight, S. (Eds.), Proceedings 11th Interaction and Concurrency Experience, ICE 2018, Madrid, Spain, June 20-21, 2018. In: EPTCS, vol. 279, pp. 52–67. http://dx.doi.org/10.4204/EPTCS.279.7.

Kamburjan, E., Hähnle, R., Schön, S., 2018. Formal modeling and analysis of railway operations with active objects. Sci. Comput. Program. 166, 167–193. http://dx.doi.org/10.1016/j.scico.2018.07.001.

Kästner, C., Apel, S., Thüm, T., Saake, G., 2012a. Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. 21 (3), http://dx.doi.org/10.1145/2211616.2211617.

Kästner, C., Ostermann, K., Erdweg, S., 2012b. A variability-aware module system. In: Leavens, G.T., Dwyer, M.B. (Eds.), Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '12, ACM, New York, NY, USA, pp. 773–792. http://dx.doi.org/10.1145/2384616.2384673.

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F., 2014. DeltaJ 1.5: delta-oriented programming for Java 1.5. In: Kolodziej, J., Childers, B.R. (Eds.), Intl. Conf. on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ, Cracow, Poland. ACM, pp. 63–74. http://dx.doi.org/10.1145/2647508.

Lagorio, G., Servetto, M., Zucca, E., 2009. Featherweight Jigsaw: A minimal core calculus for modular composition of classes. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming. Springer-Verlag, Berlin, Heidelberg, pp. 244–268. http://dx.doi.org/10.1007/978-3-642-03013-0_12.

Lagorio, G., Servetto, M., Zucca, E., 2012. Featherweight Jigsaw - Replacing inheritance by composition in Java-like languages. Inform. and Comput. 214, 86–111. http://dx.doi.org/10.1016/j.ic.2012.02.004.

Lienhardt, M., Clarke, D., 2012. Conflict detection in delta-oriented programming. In: ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. In: LNCS, vol. 7609, Springer, pp. 178–192. http://dx.doi.org/10.1007/978-3-642-34026-0_14.

Liquori, L., Spiwack, A., 2008. Feathertrait: A modest extension of featherweight java. ACM Trans. Program. Lang. Syst. 30 (2), 11:1–11:32. http://dx.doi.org/10.1145/1330017.1330022.

Mauliadi, R., Setyautami, M.R.A., Afriyanti, I., Azurat, A., 2017. A platform for charities system generation with SPL approach. In: Proc. Intl. Conf. on Information Technology Systems and Innovation. ICITSI, IEEE, New York, NY, USA, pp. 108–113. http://dx.doi.org/10.1109/ICITSI.2017.8267927.

Mikhajlov, L., Sekerinski, E., 1998. A study of the fragile base class problem. In: ECOOP'98. In: LNCS, vol. 1445, pp. 355–383, cited By 1.

Nierstrasz, O., Ducasse, S., Schärli, N., 2006. Flattening traits. J. Object Technol. 5 (4), 129–148. http://dx.doi.org/10.5381/jot.2006.5.4.a4.

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

Reppy, J.H., Turon, A., 2007. Metaprogramming with traits. In: Ernst, E. (Ed.), ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. In: Lecture Notes in Computer Science, vol. 4609, Springer, pp. 373–398. http://dx.doi.org/10.1007/978-3-540-73589-2_18.

Roffe, A.J., Calderon, J.S.T., 2021. Random formula generators. CoRR abs/2110.09228. URL: https://arxiv.org/abs/2110.09228. arXiv:2110.09228.

Roman, S., 2008. Lattices and Ordered Sets. Springer New York, URL: https://books.google.it/books?id=NZN8aum26LgC.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines. In: Software Product Lines: Going beyond (SPLC 2010). In: LNCS, vol. 6287, pp. 77–91. http://dx.doi.org/10.1007/978-3-642-15579-6_6.

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

Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K., 2012. Software diversity. Int. J. Softw. Tools Technol. Transf. 14 (5), 477–495. http://dx.doi.org/10.1007/s10009-012-0253-y.

Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P., 2003. Traits: Composable units of behaviour. In: Cardelli, L. (Ed.), ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings. In: LNCS, vol. 2743, Springer, pp. 248–274. http://dx.doi.org/10.1007/978-3-540-45070-2_12.

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

Schröter, R., Siegmund, N., Thüm, T., 2013a. Towards modular analysis of multi product lines. In: Proceedings of the 17th International Software Product Line Conference Co-Located Workshops. SPLC '13, ACM, pp. 96–99. http://dx.doi.org/10.1145/2499777.2500719.

Schröter, R., Thüm, T., Siegmund, N., Saake, G., 2013b. 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. pp. 9:1–9:5. http://dx.doi.org/10.1145/2430502.2430515.

Setyautami, M.R.A., Adianto, D., Azurat, A., 2018. Modeling multi software product lines using UML. In: Proc. 22nd Intl. Systems and Software Product Line Conference, Vol. 1. ACM, New York, NY, USA, pp. 274–278. http://dx.doi.org/10.1145/3233027.3236400.

Setyautami, M.R.A., Hähnle, R., 2021. An architectural pattern to realize multi software product lines in Java. In: Grünbacher, P., Seidl, C., Dhungana, D., Lovasz-Bukvova, H. (Eds.), Proc. 15th Intl. Working Conf. on Variability Modelling of Software-Intensive Systems, Krems, Austria. ACM Press, pp. 9:1–9:9. http://dx.doi.org/10.1145/3442391.3442401.

Setyautami, M.R.A., Rubiantoro, R.R., Azurat, A., 2019. Model-driven engineering for delta-oriented software product lines. In: 26th Asia-Pacific Software Engineering Conf., APSEC, Putrajaya, Malaysia. IEEE, pp. 371–377. http://dx.doi.org/10.1109/APSEC48747.2019.00057.

Smith, C., Drossopoulou, S., 2005. Chai: Traits for java-like languages. In: Black, A.P. (Ed.), ECOOP 2005 - Object-Oriented Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 453–478. http://dx.doi.org/10.1007/11531142_20.

Strniša, R., Sewell, P., Parkinson, M., 2007. The Java module system: Core design and semantic definition. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '07, Association for Computing Machinery, New York, NY, USA, pp. 499–514. http://dx.doi.org/10.1145/1297027.1297064.

Thaker, S., Batory, D., Kitchin, D., Cook, W., 2007. Safe composition of product lines. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering. GPCE '07, ACM, New York, NY, USA, pp. 95–104. http://dx.doi.org/10.1145/1289971.1289989.

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

Trujillo-Tzanahua, G.I., Juárez-Martínez, U., Aguilar-Lasserre, A.A., Cortés-Verdín, M.K., 2018. Multiple software product lines: applications and challenges. In: Mejia, J., Muñoz, M., Rocha, A., Quiñonez, Y., Calvo-Manzano, J. (Eds.), Trends and Applications in Software Engineering. Springer International Publishing, Cham, pp. 117–126. http://dx.doi.org/10.1007/978-3-319-69341-5_11.

Wirth, N., 1980. The module: A system structuring facility in high-level programming languages. In: Tobias, J.M. (Ed.), Language Design and Programming Methodology. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–24. http://dx.doi.org/10.1007/3-540-09745-7_1.

Wong, P.Y.H., Diakov, N., Schaefer, I., 2012. Modelling distributed adaptable object oriented systems using HATS approach: A fredhopper case study (invited paper). In: Beckert, B., Damiani, F., Gurov, D. (Eds.), 2nd Intl. Conf. on Formal Verification of Object-Oriented Software, Torino, Italy. In: LNCS, vol. 7421, Springer, pp. 49–66. http://dx.doi.org/10.1007/978-3-642-31762-0_5.