Variability Modules for ABS — @ ABS 2021

(see “Variability Modules for Java-like Languages” @ SPLC 2021)

Ferruccio Damiani¹, Reiner Hähnle², Eduard Kamburjan³, Michael Lienhardt⁴, Luca Paolini¹

¹University of Turin
²TU Darmstadt
³University of Oslo
⁴ONERA
A Product Line for Railway Signals

Form

Light
A Multi-Product Line for Railway Stations (involving Signals, Switches,...)
### Challenges

- **Interoperability:** Multiple variants from one PL must coexist and be interoperable: each variant is *encapsulated* and multiple variants *share common code* and may depend on each other.

- **Checkable:** Dependencies between multiple PLs and their variants must be easily trackable.

- **Naturalness:** The mechanism must be natural to the user.
Interoperable Variants

Challenges

- **Interoperability**: Multiple variants from one PL must coexist and be interoperable: each variant is *encapsulated* and multiple variants *share common code* and may depend on each other.

- **Checkable**: Dependencies between multiple PLs and their variants must be easily trackable.

- **Naturalness**: The mechanism must be natural to the user.

Variability Modules

Adopt the *module* mechanism to structure the variability and manage dependencies.
Modules, Delta- and Object-Oriented Programming in ABS without VMs
Modules

```java
module M;
export I;
import * from M2;

interface I { I m(); }
class C implements I { I m() { return new C(); } }
```
Modules

module M;
export I;
import * from M2;

interface I { I m(); }
class C implements I { I m() { return new C(); }

Modules

Modules introduce an encapsulated scope for classes and interfaces
- Classes and interfaces may be exported and imported.
- References can be unqualified (I) or qualified (M.I, M2.I)
Variability in ABS is based on deltas and features:

- Deltas are sets of modifications of classes and interfaces
- Features are restricted by a feature model
- Each delta is activated by a propositional formula over features
- A set of features is a \textit{configuration}
- A set of features adhering to the feature model is a \textit{product}
Variability in ABS is based on deltas and features:

- Deltas are sets of modifications of classes and interfaces
- Features are restricted by a feature model
- Each delta is activated by a propositional formula over features
- A set of features is a *configuration*
- A set of features adhering to the feature model is a *product*

```plaintext
features Light, Form, Dir with Light <-> !Form;
delta LDelta;
adds class Signals.CBulb { };
modifies class Signals.CSig { Unit addBulb() { new CBulb();} };
delta LDelta when Light;
```
Variability Modules (VMs) — they can be added on top of any Java-like language
## Variability Modules (VMs)

A VM is a module with a variability model local to its elements.
- Each delta can only modify elements within its module
- Two VMs cannot share features

Non-variable elements must be annotated with the `unique` modifier.

## Referencing Variants

To reference a variant of a non-`unique` element in a VM, the reference must be annotated with the `product`.

```plaintext
product P = { F2 };  
I v = new C() with { F1 };  
v = new C() with P;  
v = new C() with P + { F1 };  
```
// MODULE HEADER
module Signals; export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
product LSig = {Light};

// CORE PART
unique interface ISig { Bool eqAspect(ISig); Unit setToHalt(); } 
class CSig implements ISig { }

// DELTA PART
delta LDelta;
  adds class CBulb { }
  modifies class CSig { Unit addBulb() { new CBulb();} }
delta FDelta; modifies class CSig { }
...
delta LDelta when Light;
...
module Signals; export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
product LSig = {Light};

unique interface ISig { Bool eqAspect(ISig); Unit setToHalt(); } 
class CSig implements ISig { }

// DELTA PART
delta LDelta;
    adds class CBulb { }; 
    modifies class CSig { Unit addBulb() { new CBulb();} }; 
delta FDelta; modifies class CSig { }; 
... 
delta LDelta when Light; 
...
// MODULE HEADER
module Signals; export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
product LSig = {Light};

// CORE PART
unique interface ISig { Bool eqAspect(ISig); Unit setToHalt(); } 
class CSig implements ISig { }

// DELTA PART
delta LDelta;
  adds class CBulb { }
  modifies class CSig { Unit addBulb() { new CBulb();} }
delta FDelta; modifies class CSig { }
... 
delta LDelta when Light;
...
// MODULE HEADER
module Signals; export LSig, CSig, ISig;
features Light, Form, Dir with Light <-> !Form;
product LSig = {Light};

// CORE PART
unique interface ISig { Bool eqAspect(ISig); Unit setToHalt(); } 
class CSig implements ISig { }

// DELTA PART
delta LDelta;
   adds class CBulb { };
   modifies class CSig { Unit addBulb() { new CBulb(); } };

... 
delta LDelta when Light;
...
// MODULE HEADER
module InterlockingSys; import * from Signals; import * from Switches;
features Modern, DirOut with True;

product PS for Switches = { Modern => {Electric}, !Modern => {Mechanic} }
...

// CORE PART
unique interface IILS { }
class CILS {
  Bool testSig() {
    ...
    ISig sigNormal = new CSig() with LSig;
    ISig sigShunt = new CSig() with {Form};
    return sigNormal.eqAspect(sigShunt);
  }
  ISwitch createSwitch() { return new CSwitch() with PS; }
  ...
}
// MODULE HEADER
module InterlockingSys; import * from Signals; import * from Switches;
features Modern, DirOut with True;

product PS for Switches = { Modern => {Electric}, !Modern => {Mechanic} }
...

// CORE PART
unique interface IILS { }

class CILS {
  Bool testSig() {
    ...
    ISig sigNormal = new CSig() with LSig;
    ISig sigShunt  = new CSig() with {Form};
    return sigNormal.eqAspect(sigShunt);
  }
  ISwitch createSwitch() { return new CSwitch() with PS; }
  ...
}
// MODULE HEADER
module InterlockingSys; import * from Signals; import * from Switches; features Modern, DirOut with True;

product PS for Switches = { Modern => {Electric}, !Modern => {Mechanic} }

// CORE PART
unique interface IILS { }
class CILS {
  Bool testSig() {
    ...
    ISig sigNormal = new CSig() with LSig;
    ISig sigShunt = new CSig() with {Form};
    return sigNormal.eqAspect(sigShunt);
  }
  ISwitch createSwitch() { return new CSwitch() with PS; }
  ...
}
// MODULE HEADER
module InterlockingSys; import * from Signals; import * from Switches;
features Modern, DirOut with True;

product PS for Switches = { Modern => {Electric}, !Modern => {Mechanic} }
...

// CORE PART
unique interface IILS { }
class CILS {
  Bool testSig() {

    ISig sigNormal = new CSig() with LSig;
    ISig sigShunt  = new CSig() with {Form};
    return sigNormal.eqAspect(sigShunt);
  }

  ISwitch createSwitch() { return new CSwitch() with PS; }
  ...
}
Example

// MODULE HEADER
module InterlockingSys; import * from Signals; import * from Switches;
features Modern, DirOut with True;

product PS for Switches = { Modern => {Electric}, !Modern => {Mechanic} } ...

// CORE PART
unique interface IILS { }
class CILS {
    Bool testSig() {
        ...
        ISig sigNormal = new CSig() with LSig;
        ISig sigShunt = new CSig() with {Form};
        return sigNormal.eqAspect(sigShunt);
    }
    ISwitch createSwitch() { return new CSwitch() with PS; }
}

Damiani, Hähnle, Kamburjan, Lienhardt, Paolini — Variability Modules for ABS — © ABS 2021
Flattening
A program with VMs can be syntactically flattened into a program without VMs. Further compilation steps are performed on the flattened program.

- While there are references to VM $M$ with product $P$ on $E$
  1. If a module $M_P$ exists, go to 6.
  2. Make a copy $M_P$ of $M$
  3. Delete all unique elements from $M_P$
  4. Add import clause for $M$
  5. Apply deltas according to $P$ to $M_P$
  6. Replace reference with qualified name $M_P.E$
  7. Adapt import clause of dependent module

- For each VM $M$, delete all non-unique elements.
- Delete all product declarations and deltas
module Signals;
export ISig;
interface ISig { Bool eqAspect(ISig); Unit setToHalt(); }

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

module InterlockingSys;
import * from Signals; import * from Signals_Light;
...

class CILS {
  Bool testSig() {
    ...
    ISig sigNormal = new Signals_Light.CSig();
    ISig sigShunt  = new Signals_Form.CSig();
    return sigNormal.eqAspect(sigShunt);
  }
  ...
}
module Signals;
export ISig;
interface ISig { Bool eqAspect(ISig); Unit setToHalt(); }

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

module InterlockingSys;
import * from Signals; import * from Signals_Light;
...

class CILS {
Bool testSig() {
...
   ISig sigNormal = new Signals_Light.CSig();
   ISig sigShunt  = new Signals_Form.CSig();
   return sigNormal.eqAspect(sigShunt);
}
...
}
module Signals;
export ISig;
interface ISig { Bool eqAspect(ISig); Unit setToHalt(); }

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

module InterlockingSys;
import * from Signals; import * from Signals_Light;
...

class CILS {
    Bool testSig() {
        ...
        ISig sigNormal = new Signals_Light.CSig();
        ISig sigShunt   = new Signals_Form.CSig();
        return sigNormal.eqAspect(sigShunt);
    }
    ...
}
module Signals;
export ISig;
interface ISig { Bool eqAspect(ISig); Unit setToHalt(); }

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

module InterlockingSys;
import * from Signals; import * from Signals_Light;
...

class CILS {
Bool testSig() {
...
ISig sigNormal = new Signals_Light.CSig();
ISig sigShunt = new Signals_Form.CSig();
return sigNormal.eqAspect(sigShunt);
...
}
Principle of Encapsulated Variability

**Definition: Principle of Encapsulated Variability (PEV)**

A module depends on other variable modules only via *unique* elements or elements with a specified variant.

**Checking Uniqueness**

Each unique element can only depend on *unique* elements or elements with a specified variant *within its own module*. A delta can only modify non-*unique* classes.

**Theorem**

*If a program adheres to the PEV and all the delta-applications in each VM succeed, then flattening succeeds.*
Examples

**Invalid dependency on M.D**

```java
module M; features ...;

class D implements I {}
...

module N;
...

class E { Unit m() { ... new D(); } }
```

**Invalid internal reference on M.D**

```java
module M;
...
unique interface I {}
unique class C { I m(){ return new D(); }}
class D implements I {} 
...
delta Del;
modifies class D;
```
Discussion and Evaluation
Discussion

Implementation

Implementation for ABS available at

https://github.com/abstools/abstools/tree/local_productlines

- Framework and PEV not specific for ABS
- Far more usable than prior approach to interoperability
  - Extends concepts natural to the programmer
  - Introduces no new scopes
- Not discussed here: formal semantics, open product declarations, exact handling of import/export clauses
**Evaluation**

**Case Study 1 (AISCO): AVS-VM vs. External Tool Chain**
- Modular web portal using ABS for variability of modules.
- Used an external tool chain to implement interoperability, could be re-implemented completely using AVS-VM.

**Case Study 2 (Railway Operations): AVS-VM vs. Traits**
ABS model of railway operations of Deutsche Bahn. Infrastructure used object-orientation and traits for interoperable variability.
- Using AVS-VM made implicit constraints (certain traits cannot be used together) explicit in the feature model.
- -25% LoC for relevant infrastructure modeling
### Case Study 3 (Memory Models): AVS-VM vs. ABS SPL

Extend ABS model of weak memory models to handle an architecture with 4 different memory models
Case Study 3 (Memory Models): AVS-VM vs. ABS SPL

Extend ABS model of weak memory models to handle an architecture with 4 different memory models.
Case Study 3 (Memory Models): AVS-VM vs. ABS SPL

Extend ABS model of weak memory models to handle an architecture with 4 different memory models.
Case Study 3 (Memory Models): AVS-VM vs. ABS SPL

Extend ABS model of weak memory models to handle an architecture with 4 different memory models

- Naïve approach: copy variability, then refactor common part
- AVS-VM: -63% LoC, scales for any number of memory models
Conclusion

Summary

- Modules as units of variability
- Encapsulated variability for interoperable variants
- On-Going Work: Family-Based Type Checking
Summary

- Modules as units of variability
- Encapsulated variability for interoperable variants
- On-Going Work: Family-Based Type Checking

Thanks you for your attention!