Variability Modules for Java-like Languages
Artifact Submission

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

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

1 Variability Modules

This artifact accompanies an SPLC article [2], which will be available under https://doi.org/10.1145/3461001.3471143. For full details, we refer to that article, the following is a summary of its main findings.

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

Variability modules are based on the standard concept of a module, used to structure large software systems since the 1970s, as a baseline. Software modules are supported in many programming and modeling languages, including ABS, Ada, Haskell, Java, Scala, to name just a few. Because modules are intended to facilitate interoperability and encapsulation, no further *ad hoc* concepts are needed for this purpose. We merely *add variability* to modules, rendering each module a product line of standard, variability-free modules.

The main advantage of VMs is their conceptual simplicity: as a straightforward extension of standard software modules, they are intuitive to use for anyone familiar with modules and with software product lines. Each VM is both, a module and a product line of modules. This reduction of concepts not only drastically simplifies syntax, but reduces the cognitive burden on the modeler. We substantiate this claim with several case studies that compare VMs with alternative modeling approaches.

By using modules, we can deal with interoperable variants, i.e., the situation where different product variants from the same product line need to co-exist in the same context and must be interoperable [1].

For example, using an industrial case study from the literature [5], performed for Deutsche Bahn Netz AG, consider the case of railway infrastructure, where:

1. Several interdependent product lines for networks, signals, switches, etc., occur.
2. Mechanic and electric rail switches (and other components) are different variants of the same product line. Some train stations include both variants at the same time and these must be interoperable.

VMs handle different variants of the same product line by encapsulating each in its own module and carefully managing the links between them.

We formulate the VM concept as an extension of the standard concept of a module for Java-like (i.e., object-oriented, class-based and strongly typed) languages. To support variability, VMs employ *delta-oriented programming* (DOP) [7]. Specifically, our framework encompasses:

1. A theoretical foundation of VMs, including formal syntax and semantics, in terms of a core calculus.
2. An implementation of VMs as an extension of the ABS language tool chain [3,4].

We choose ABS, because it features native implementations of DOP and it was successfully used in industrial case studies for variability modeling [5,6,8]. We stress that VMs can be added on top of *any* Java-like language.

The provided case studies show that if variants need to co-exist, then VMs can reduce code size (and code duplication) when compared to an approach with monolithic product lines. A comparison of two weak memory model models in ABS (Section 8.2.3 in [2]) uses 272% more code without VMs, as modules have to be duplicated. Similarly, a remodeling of railway infrastructure with VMs uses ∼25% less code than the previous version based on traits (Section 8.2.2 in [2]). The final case study (Section 8.2.1 in [2]) compares a VM model with a system that used an external tool chain to mimic VMs.

2 Context

Relation to ABS and Code Availability. The artifact is a version of the ABS compiler, extended with the variability module system described above, as well as several examples to reproduce the case studies in Section 8 of the main article. The extension is available online under the following URL (note that the code is in the `variable_mod` branch, not the master branch).

`https://github.com/Edkamb/abstools/tree/variable_mod`

Variability modules will soon be merged into the main branch of the ABS compiler once an ongoing refactoring phase in the main branch will be completed. The documentation of the pull request is located at

`https://github.com/abstools/abstools/pull/279`

ABS Language documentation can be found at `https://abs-models.org/manual/`, the ticketing system to report problems at `https://github.com/abstools/abstools/issues`.

Changed Code. In our VM implementation we tried to reuse as much code as possible from the existing variability system (based on deltas and traits) of ABS. There are no changes to the code implementing delta application. The changes to the frontend are made to the grammar[5] and the AST description language[6]. The unfolding mechanism itself is implemented in the `ProductFlattener` aspect [7] and in related aspects[8] (error reporting and adaptation of the type system to accommodate the new constructs). Finally, the integration into the workflow is implemented as part of the main method in `frontend/src/main/java/org/abs_models/frontend/parser/Main.java`. Tests are supplied at `frontend/src/test/java/org/abs_models/frontend/delta/localpls/LocalPLsTest.java`.

3 Instructions

First, VirtualBox 6.1 has to be installed in your system. It can be downloaded from:

`https://www.virtualbox.org/wiki/Downloads`

A virtual machine with the extended ABS compiler preinstalled is available under

`https://zenodo.org/record/5042218`

[5] `src/main/java/org/abs_models/xtext/Abs.xtext`
[6] `frontend/src/main/java/org/abs_models/frontend/ast/*`
[7] `frontend/src/main/java/org/abs_models/frontend/delta/ProductFlattener.jadd`
[8] `frontend/src/main/java/org/abs_models/frontend/delta/*`

The password for the virtual machine is `variable`, the user is `abs`. There is no special requirement on the settings of the VM, it was tested with virtualbox and default settings.

The virtual machine can be loaded in VirtualBox by clicking in the file "VarModVM.ova" dowloaded from the above zenodo repos. This file and the preprint of the SPLC companion paper in PDF can be fund in the desktop of the virtual machine. They can be read in the VM by left-clicking on them and selecting in the contextual-menu "Open with other applications" and by selecting Firefox Web Browser.

To run commands, right-click on the opened directory and select `Open Terminal Here`. In the terminal you can run the code, open the `compiler/abstools` directory in the desktop of the VM and run the commands described in Section 3.1. The generation processes produce a number of warnings, which can all be safely ignored.

3.1 Examples

The conference paper [2] describes several case studies that compare variability modules with other ways to model similar situations. These can be reproduced with the help of the following commands:

- To compile the running example (Section 2), run
  `java -jar frontend/build/libs/absfrontend.jar --prettyprint examples/VM/Rails.abs`
- To compile the AISCO portal (Section 8.2.1, comparing with an external tool chain), run
  `java -jar frontend/build/libs/absfrontend.jar --prettyprint examples/VM/Total.abs`
- To compile the FormbaR model refactoring (Section 8.2.2, comparing with traits), run
  `java -jar frontend/build/libs/absfrontend.jar --prettyprint examples/VM/formbar/POSTVM/*abs`
  The pre-refactoring model is in `examples/VM/formbar/PREVM`
- To compile the Memory model (Section 8.2.3, comparing with global product lines), run
  `java -jar frontend/build/libs/absfrontend.jar --prettyprint examples/Memory/Mem_VM.abs`
  The model without VMs is in `examples/Memory/VM/Mem_no_VM.abs`

3.2 Compilation

To recompile run

```
make
```

from the root directory of the compiler (with `MAKEFILE` in it). The build process produces several warnings. These also indicate no errors and can be ignored. The resulting `.jar` file is generated in `compiler/abstools/frontend/build/libs/`. Observe that the name of the generated `.jar` file includes a suffix depending on the current version, for example,

```
absfrontend-variable_mod-old-parser-sunset-1767-g61560577f.jar
```

To clone the whole repository, run

```
git clone https://github.com/Edkamb/abstools.git .
git checkout variable_mod
```

3.3 Setting up the VM

To set up the VM, generate a new virtual machine running a fresh install of Xubuntu 21.04 and run the following to install the dependencies.

```
sudo apt install git openjdk-8-jdk erlang make
```

Afterwards, follow the steps in Section 3.2. The provided VM was generated and tested using VirtualBox (`https://www.virtualbox.org/wiki/Downloads`) 6.1 with the default settings.

References

1. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. Interoperability of software product line variants. In *SPLC*, pages 264–268. ACM, 2018.
2. F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, and L. Paolini. Variability modules for java-like languages. In *SPLC*. ACM, 2021.
3. R. Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In M. Bonsangue, F. de Boer, E. Giachino, and R. Hähnle, editors, *Intl. School on Formal Models for Components and Objects: Post Proceedings*, volume 7866 of *LNCS*, pages 1–37. Springer, 2013.
4. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, pages 142–164, 2010.
5. E. Kamburjan, R. Hähnle, and S. Schön. Formal modeling and analysis of railway operations with Active Objects. *Science of Computer Programming*, 166:167–193, Nov. 2018.
6. R. Mauliadi, M. R. A. Setyautami, I. Afriyanti, and A. Azurat. A platform for charities system generation with SPL approach. In *Proc. Intl. Conf. on Information Technology Systems and Innovation (ICITSI)*, pages 108–113, New York, NY, USA, 2017. IEEE.
7. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (SPLC 2010)*, volume 6287 of *LNCS*, pages 77–91, 2010.
8. P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study (invited paper). In *2nd International Conference on Formal Verification of Object-Oriented Software, Torino, Italy*, volume 7421 of *LNCS*. Springer, 2012.