Interoperability of Software Product Line Variants

Ferruccio Damiani
University of Torino
Torino, Italy
ferruccio.damiani@unito.it

Reiner Hähnle
Technische Universität Darmstadt
Darmstadt, Germany
haehnle@cs.tu-darmstadt.de

Eduard Kamburjan
Technische Universität Darmstadt
Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Michael Lienhardt
University of Torino
Torino, Italy
mlienhardt@di.unito.it

ABSTRACT

Software Product Lines are an established mechanism to describe multiple variants of one software product. Current approaches however, do not offer a mechanism to support the use of multiple variants from one product line in the same application. We experienced the need for such a mechanism in an industry project with German Railways where we do not merely model a highly variable system, but a system with highly variable subsystems. We present the design challenges that arise when software product lines have to support the use of multiple variants in the same application, in particular: How to reference multiple variants, how to manage multiple variants to avoid name clashes, and how to keep multiple variants interoperable.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Reusability**; **Software product lines**; *Software prototyping*; • **Applied computing** → Engineering;

ACM Reference Format:
Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. 2018. Interoperability of Software Product Line Variants . In *22nd International Systems and Software Product Line Conference - Volume A (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3233027.3236401

1 MOTIVATION

The challenge case we pose here is motivated by an industrial project with German Railways (DB Netz AG) [9] that includes a detailed, executable model of railway infrastructure.

Rail networks consist of a large number of technical components, including signals, switches, tracks, magnets, axle counters, and so on. Some kinds of components show a high degree of variability. For example, signals can be based on light or on shape, they can be arranged in different ways, they can be controlled manually, mechanically, electrically, or digitally, etc. Dozens, perhaps hundreds, of different types of railway components exist. To manage this variability, it is advisable to arrange components as a product line. For example, product line `SignalLine` is used to produce different variants of railway signals, while product line `SwitchLine` is used for switches. Each of these product lines may have twenty or more features (and comprise hundreds of products), but to keep things simple, we use merely two features and two products per product line in our example: a signal is either a main or a pre-signal (the latter announces the signal aspect of the main signal to the train driver), a switch is either an electric switch or a manual switch. Following the syntax of [3], we can represent these product lines as follows:

```
productline SignalLine;
  features Main, Pre;
  product MainSignal(Main);
  product PreSignal(Pre);

productline SwitchLine;
  features Electric, Manual;
  product ESwitch(Electric);
  product MSwitch(Manual);
```

In this syntax, **productline** starts the declaration of a new product line; **features** declares the features of the product line; and **product** declares a product of the product line. For example, **product** `MainSignal(Main)` declares the product `MainSignal` and states that it corresponds to the selection of the feature `Main`.

Stations consist of several signals and switches (plus other elements, such as straight line tracks, platforms, bumpers, etc., which we ignore here). Again, there are many station types (terminals, shunting stations, ...), so it makes sense to arrange stations as a product line as well, say, `StationLine`. But now we face a problem: a station does not merely comprise switches as well as signals, but typically it also contains *different* types of signals and switches. Hence we need to address how a variant of `StationLine` can make use of *multiple product variants*.

Existing product line models and approaches seem to address the generation of a single product variant. Our use case requires to manage the interoperability between *multiple* product variants from one or more product lines within a *single* application.

Indeed, the challenge case presented here shifts the focus of product line-based development from *highly variable systems* to systems

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3236401

with *highly variable subsystems*. As our case study shows, multiple subsystems and, hence, multiple product lines arise naturally in real world systems. As a consequence, the management of multiple variants from *one* product line must interface with an environment that comprises further, multiple product lines.

The challenge proposed here is the question of how to handle multiple variants of one product line. This must be solved before the natural next step, multiple variants from different interdependent product lines can be taken. Hence, we discuss Multi Software Product Lines (MPLs) [11, 14, 17] only from the perspective of their future extension relative to the challenge presented here.

Our perspective entails that it is not sufficient to look at the variability aspect when designing product lines. Rather, it is as well necessary to address the question of *modularity*. This is conventionally done at the granularity of architectural components such as interfaces, classes, traits, etc. Our case study shows that, whenever a product line is responsible for a clearly defined subsystem, we must also have concepts that allow to manage its relation to other sub-components within one and the same model. In other words, it is necessary to *lift* architectural issues around modularity and interoperability from the level of individual components to the level of product lines. This leads to a number of *design challenges* that we explain in greater detail in the following sections:

- How different variants (and their content) from a product line can be *accessed* and can *interoperate* within the same application (possibly being a variant of another product line)?
- How do product lines *expose* their variants to other applications and product lines?
- How to resolve *name clashes* when two variants declare the same element (such as a class or an interface), and how these elements can soundly *interact* w.r.t. the type system?

In the following sections, we illustrate the design challenges with examples written in the ABS language [12]: ABS is an Object-Oriented language with a syntax close to Java and is used to implement our challenge case [9]. However, we claim that the design challenges discussed in this article are not specific to any product line implementation approach. They arise as soon as one attempts to combine product lines and variants of the same product line.

2 REFERENCING PRODUCT VARIANTS

The usage of multiple products requires that a product line can be referenced and configured from outside. This raises the question of how specific variants of classes or interfaces that realize different products may be referred to. We present some considerations around this issue, using an *ad hoc* syntax. Assume that a product line for stations uses the following interfaces for signals and switches, without specifying implementing classes:

```
interface Sig { ... };
interface Sw {... };
```

Further, assume product line `SignalLine` declares a class `Signal` that implements `Sig` and `SwitchLine` declares a class `Switch` that implements `Sw`, respectively. Our task is to model a station with two signals, one pre-signal and one main signal. We discuss two possible approaches for referencing class implementations declared in different product variants.

2.1 Product Variant References at Types

One possiblity to syntactically reference a specific variant of class `Signal` would be analogous to the use of traits in Scala [15].[1]

Example 2.1. The following method of a station model adds a pre-signal as well as a main signal and registers both as covering signals to a switch.

```
Unit constructStation(Sw switch) {
  Sig s1 = new Signal()   from SignalLine.PreSignal;
  Sig s2 = new Signal(s1) from SignalLine.MainSignal;

  switch.coveredBy(s1);
  switch.coveredBy(s2);
}
```

The intended semantics of "`new C(...) from pl.prod`" is that `C` is the class implementation provided by product variant `prod` of product line `pl`.

The example shows how object creation might reference classes declared in different product variants. However, it may also be necessary to reference interfaces and types: for example, when interface `Sig` is not declared in the station product line, but in `SignalLine`, then it must be annotated with the specific variant of `Sig` that is to be used, as shown here:

```
Sig [from SignalLine.PreSignal] s1 =
    new Signal() from SignalLine.PreSignal;
```

We expect that an approach that integrates product line configuration into type references is a natural extension of the object-oriented paradigm, but it might be very verbose.

2.2 Product Variant References at Larger Scopes

Instead of referencing variants at the level of types, one may reference them at the level of components with a larger scope: method, class, package, etc. Consider the following method that constructs a station with multiple main signals sharing the same pre-signal: several references at the type-level are replaced by a default reference given in the method signature:

```
Unit constructStation() with Signal
                       from SignalLine.MainSignal {
  Sig s1 = new Signal() from SignalLine.PreSignal;
  Sig s2 = new Signal(s1);
  Sig s3 = new Signal(s1);
  Sig s4 = new Signal(s1); ...
}
```

This approach reduces the verbosity when referencing variants. Another idea is to employ principles from Multi Product Lines [11]: here one *limits* the variants that can possibly be referenced, instead of (or in addition to) listing them explicitly. Consider the following example, where "`forces SignalLine.Pre`" has the intended semantics that all referenced variants of `SignalLine` *must* have the feature `Pre`. A reference to `SignalLine.MainSignal` would raise an error.

[1] Recall that Scala traits are in fact mixins.

The following method adds pre-signals and uses the **forces** mechanism to ensure that it is impossible to reference a main signal inside the method body.

```
Unit constructPreSigs() forces SignalLine.Pre {
  sig1 = new Signal() from SignalLine.PreSignal;
  sig2 = new Signal() from SignalLine.PreSignal;
}
```

3 EXPOSING VARIANTS

The standard interface of a product line to the outside world is a set of products or a set of features. When composing variants from multiple products or product lines, however, classes and interfaces inside a product variant have to be referenced. When product lines are used in this manner, i.e. to model *subsystems*, should every declared, added or modified element be referencable from the outside? Consider again the `SignalLine` example in Section 2.1 that declares a `Signal` class. Assume it also declares as well a subsidiary signal[2] which is only added to main signals under certain circumstances as class `SubsidiarySignal`: should it be possible to reference `SubsidiarySignal` as follows from `StationLine`?

```
new SubsidiarySignal from SignalLine.MainSignal;
```

This also raises the question how to deal with the case when a class from a specific variant is referenced, but is not present or modified in that variant. For example, the `SubsidiarySignal` class is usually not added to `SignalLine.PreSignal`. What should then be the semantics of the following?

```
new SubsidiarySignal from SignalLine.PreSignal;
```

An obvious solution would be, similar as in a module declaration, to equip the interface of a product line or a product with an export list of classes and interfaces being visible to the outside:

```
productline SignalLine exposes Signal;
  features Main, Pre;
  product MainSignal(Main) exposes SubsidiarySignal;
  product PreSignal(Pre);
```

Finally, there is the question of what to do with classes and interfaces that are provided in a product variant, but not needed, i.e. referenced. In Example 2.1, the product variants `PreSignal` and `MainSignal` might declare many other classes in addition to `Signal`, but the latter is the only one being referenced. Are non-referenced classes and interfaces always generated? If not, what is the mechanism to detect whether they must be generated?

4 CO-EXISTENCE AND INTEROPERABILITY OF PRODUCT VARIANTS

In Example 2.1, both of the products `SignalLine.PreSignal` and `SignalLine.MainSignal` declare a class called `Signal`, and so putting these two products together will result in a name clash error, as `Signal` would be declared twice. Implicitly overriding one by the other, like it is done in mixins [1, 8], is unacceptable, as they both contain useful and different functionalities. Hence a reasonable solution would be to put the two products into their own namespace. But how to choose the namespace of each product is an open question. A first possibility is to have the developer explicitly state what the namespace is. This could be done in the declaration of the product line itself, as in the following example:

```
productline SignalLine exposes Signal;
  features Main, Pre;
  product MainSignal(Main) with namespace Main;
  product PreSignal(Pre)   with namespace Pre;
```

However, such a solution does not solve name clashes that could occur between variants of different product lines that share the same namespace. A different solution would be to have the namespace automatically chosen. How can this be done in a way that ensures the uniqueness of the namespace and is intuitive for the user?

When multiple variants of one product line are present, they must cooperate when being composed into a common product of the outer product line: in Example 2.1 the different station products require different variants of signals and as usual, when composing expressions of a program, one has to make sure that no compile time errors (in particular, typing errors) can occur.

In Example 2.1 both the `sig1` and `sig2` objects are arguments of the `coveredBy` method, so it is necessary that their types are both compatible with the type of the `coveredBy` method's argument. This is enforced when they all have the same type, that is global and not modified in any product, as in the following example:

```
1 interface Sig { ... };
2 productline SignalLine exposes Signal implements Sig;
3   features Main, Pre;
4   product MainSignal(Main);
5   product PreSignal(Pre);
```

However, enforcing that types are global and never modified can be restrictive. In an OO setting, one can view this from the perspective of subtyping. For example, is it always desirable that different variants of classes have interface types that are subinterfaces of each other? Where and how should this be declared?

In some cases it should be possible to automatically derive a subtyping relation between different variants by analyzing their code. But without any further restrictions it is easily possible to introduce diamonds or loops into the resulting type hierarchy.

5 REQUESTED SOLUTION

5.1 Solution Requirements

A solution to the challenge described above should address the following issues:

(1) Suggest syntactic constructs for referencing multiple variants of different product lines. This should allow to use multiple product variants in one single application. In particular, different product variants from the same product line can co-exist in one application.
(2) Describe mechanisms that allow multiple variants from the same product line to interact with each other within one application.

[2] In German "Ersatzsignal".

(3) Describe how your solution relates to existing Multi Software Product Line approaches (cf. Section 6) and how it handles the exposure (export) of classes and interfaces.
(4) How can your solution be incorporated into one of the existing implementation paradigms for Software Product Lines? Which of them is the most suitable?

5.2 Evaluation Criteria

To evaluate solutions to this challenge, we provide[3] three ABS models that contain interfaces, classes, and method stubs. This means that our models are not executable (however, they are compilable with the ABS compiler available from http://abs-models.org/installation/). Similarly, a solution to the challenge does not need to be executable: a solution needs only to demonstrate how co-existence and interoperability are handled.

It is, of course, not required that a solution uses ABS. It should be straightforward to translate our models to another language.

The first model we provide is for a station with two kinds of signals, the second for a station with two kinds each for signals and switches and the last with variability on station and part level. The models do not contain a product line, but multiple classes for signals, switches and/or stations.

We expect a solution to our challenge to consist of three parts:

(1) A refactoring of each of the three models into one that uses product lines to manage variability of signals, switches and stations, respectively. You can use a product line description formalism of your choice as long as it serves as input for the next part:
(2) A systematic, rule-based approach that flattens this refactored model into something similar to the original model we provided. The solution does not need to be implemented, but it should be clear how it works in the general case.
(3) A brief textual justification how the transformation defined in the previous step addresses the issues in Section 5.1.

6 RELATED WORK

Multi Software Product Lines (MPL) [11, 14, 17] constitute an approach to structure complex and variable systems into sets of interdependent product lines that can be managed in a decentralized fashion by multiple teams and stakeholders. For instance, Kästner et al. [13] proposed a variability-aware module and interface system where variability is implemented using `#ifdef` preprocessor directives and variable linking. The notion of product line composition the authors propose resolves name clashes by merging elements with the same name if possible (or else it fails), but they do not consider the problem of interoperability among variants at the type level, or the interaction in the same code of two variants of the same product line.

MPL has also been studied in the context of Delta-Oriented Programming (DOP) [3] which is how variability in ABS is implemented. The approach in [7] implements the notion of dependencies between DOP SPL by means of **import**s. The feature model and the source code of the importing product line are deeply integrated with the feature models and the source code of the imported product lines, respectively. This extension is very flexible, but it does not enforce any boundary between different product lines, it does not discuss the interoperability of the different variants at the type level, nor does it offer means to manipulate two variants of the same product line at the same time.

On the other hand, the approach in [4] enforces boundaries between product lines by using real dependencies instead of imports. Moreover, this approach discusses the problem of strong coupling between product lines, which is beyond the scope of the challenge presented here. (We realize, of course, that coupling is an important issue that is highly relevant for our use case.) However, that work only discusses name clashes informally, stating that elements can be arbitrarily renamed; it does not address the problems of interoperability between variants at the type level, nor does it offer means to manipulate two variants of the same product line at the same time.

Similarily to [4, 7], Schröter et al. [16] informally discuss the challenges when designing an MPL, and identify several aspects that a product line should expose, in addition to its variability, in order to help product line composition. In particular, they discuss *syntactic interfaces* that constitute an API of a product line that can differ for different products, and *behavioral interfaces* that describe the correct usage of this API. The issues raised by our challenge case can be seen as an extension of the challenges these authors discussed.

Finally, to the best of our knowledge, the only approach that manages the code of several variants of the same product line together at runtime is *dynamic product lines* [2, 10]. However, such an approach, as described in [5, 6] for DOP, allows for switching from one variant to another one at runtime, but does not model the interaction and collaboration between two variants of the same product line at the same moment of execution. Hence, we consider that dynamic product lines to be orthogonal to the problem discussed here.

7 CONCLUSION

Our challenge is motivated by the application of software product lines in an industrial project. It targets the expressive power and usability of product lines. We see two major advantages that the requested mechanism for the simultaneous, combined usage of multiple product variants of one or more product lines can offer:

Modeling systems with variable subcomponents. A mechanism implementing the described challenge would permit to lift product line-based design from *variable systems* to systems *with variable subcomponents*. Our challenge case shows that this is not a degenerate case of MPL, but an orthogonal approach with permits to manage multiple products from the same product line.

Variability and Modularity. A mechanism implementing the described challenge would offer a perspective on product lines from the point of view of modularity whose importance is well recognized in programming language design and software architecture. Ultimately, this can lead to a unification of modularity concepts in architecture and for dealing with variability.

A solution to our challenge would also constitute a bottom-up approach to composition of product lines into MPL that emphasizes the modularity aspect of product lines, not variability management.

[3] http://formbar.raillab.de/en/publications-and-tools/product-lines/

We expect that the discussion of possible solutions can lead to the transfer of ideas between researchers working on modularity and software variability.

Acknowledgments

This work is supported by: the FormbaR and FormETCS projects, part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt; EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; and ICT COST Action IC1402 ARVI (www.cost-arvi.eu).

REFERENCES

[1] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. *SIGPLAN Not.* 25, 10 (Sept. 1990), 303–311. https://doi.org/10.1145/97946.97982

[2] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry. *J. Syst. Softw.* 91 (May 2014), 3–23. https://doi.org/10.1016/j.jss.2013.12.038

[3] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schafer, Jan Schäfer, Rudi Schlatte, and Peter Y. H. Wong. 2011. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Formal Methods for Eternal Networked Software Systems*, M. Bernardo and V. Issarny (Eds.). LNCS, Vol. 6659. Springer, Berlin, Heidelberg, 417–457.

[4] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2017. A Formal Model for Multi SPLs. In *Fundamentals of Software Engineering (Lecture Notes in Computer Science)*, Mehdi Dastani and Marjan Sirjani (Eds.), Vol. 10522. Springer, Cham, 67–83.

[5] Ferruccio Damiani, Luca Padovani, Ina Schaefer, and Christoph Seidl. 2018. A core calculus for dynamic delta-oriented programming. *Acta Inf.* 55, 4 (2018), 269–307. https://doi.org/10.1007/s00236-017-0293-6

[6] Ferruccio Damiani and Ina Schaefer. 2011. Dynamic Delta-oriented Programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. ACM, New York, NY, USA, Article 34, 8 pages. https://doi.org/10.1145/2019136.2019175

[7] Ferruccio Damiani, Ina Schaefer, and Tim Winkelmann. 2014. Delta-oriented Multi Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*. ACM, New York, NY, USA, 232–236.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM, New York, NY, USA, 171–183. https://doi.org/10.1145/268946.268961

[9] Reiner Hähnle and Eduard Kamburjan. 2016. Uniform Modeling of Railway Operations. In *Proc. Fifth Intl. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS) (CCIS)*, Cyrille Artho and Peter Csaba Ölveczky (Eds.), Vol. 694. Springer, Cham, 55–71.

[10] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2013. *Dynamic Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 253–260. https://doi.org/10.1007/978-3-642-36583-6_16

[11] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology* 54, 8 (2012), 828 – 852. https://doi.org/10.1016/j.infsof.2012.02.002 Special Issue: Voice of the Editorial Board.

[12] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010) (LNCS)*, Bernhard K. Aichernig, Frank de Boer, and Marcello M. Bonsangue (Eds.), Vol. 6957. Springer, Berlin, Heidelberg, 142–164.

[13] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-aware Module System. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, USA, 773–792.

[14] Charles W. Krueger. 2006. New Methods in Software Product Line Development. In *Proc. 10th Intl. Software Product Line Conference*. IEEE Computer Society, Washington, DC, USA, 95–102. http://dl.acm.org/citation.cfm?id=1158337.1158683

[15] Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala* (3rd ed.). Artima Inc., 2070 N Broadway Unit 305 Walnut Creek CA 94597 USA.

[16] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. 2013. Towards Modular Analysis of Multi Product Lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*. ACM, New York, NY, USA, 96–99.

[17] Leopoldo Teixeira, Paulo Borba, and Rohit Gheyi. 2015. Safe Evolution of Product Populations and Multi Product Lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. ACM, New York, NY, USA, 171–175. https://doi.org/10.1145/2791060.2791084