Experience in Specializing a Generic Realization Language for SPL Engineering at Airbus

Damien Foures, Mathieu Acher, Olivier Barais, Benoit Combemale, Jean-Marc Jézéquel, Jörg Kienzle

Experience in Specializing a Generic Realization Language for SPL Engineering at Airbus

Damien Foures
Airbus (Toulouse, France)

Mathieu Acher
Univ Rennes, Inria, CNRS, IRISA

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA

Benoit Combemale
Univ Rennes, Inria, CNRS, IRISA

Jean-Marc Jézéquel
Univ Rennes, Inria, CNRS, IRISA

Jörg Kienzle
McGill University / University of Málaga

***Abstract*—In software product line (SPL) engineering, feature models are the de facto standard for modeling variability. A user can derive products out of a base model by selecting features of interest. Doing it automatically, however, requires a *realization model*, which is a description of how a base model should be modified when a given feature is selected/unselected. A realization model then necessarily depends on the base metamodel, asking for *ad hoc* solutions that have flourished in recent years. In this paper, we propose *Greal*, a *generic* solution to this problem in the form of (1) a generic declarative realization language that can be automatically composed with one or more base metamodels to yield a domain-specific realization language and (2) a product derivation algorithm applying a realization model to a base model and a resolved model to yield a derived product. We describe how, on top of Greal, we specialized a realization language to support both positive and negative variability, fit the syntax and semantics of the targeted language (BPMN) and take into account modeling practices at Airbus. We report on lessons learned of applying this approach on Program Development Plans based on business process models and discuss open problems.**

I. Introduction

Software Product Line (SPL) engineering [1], [2] aims at capturing the commonalities (assumptions true for each family member) and variability (assumptions about how individual family members differ) among several software products [3]. Variability management is thus a key feature that distinguishes SPL engineering from other software development approaches [4]. It concerns the entire development life cycle, from requirements elicitation [5] and tracing [6] to product derivation [7] to product testing [8]. Modeling variability allows a project to capture and select which version of which variant of any particular component is wanted in the system. A user can then derive products out of a base model by selecting the features of interest for a specific configuration.

Doing it automatically, however, requires a *realization model*, which is a description of how a base model should be modified when a given feature is selected or unselected in a given configuration. For instance, we could want that whenever the feature f_1 is selected, then the class X would be added to a base (class) model, or the state S removed from a base (StateChart) model. A realization model then necessarily depends on the base metamodel, asking for *ad hoc* solutions that have flourished in recent years.

In this paper, we propose *Greal* a *generic* solution to this problem in the form of (1) a generic declarative realization language that can be automatically composed with one or more base metamodels to yield a domain-specific realization language and (2) a product derivation algorithm applying a realization model to a base model and a resolved model to yield a derived product. We explain our approach using the well-known use case of the *Expression Problem Product Line* (EPL), highlighting the need for both positive and negative variability even for this simple case. We then describe how, on top of *Greal*, we specialized a realization language to support both positive and negative variability, fit the syntax and semantics of the targeted language (BPMN) and take into account modeling practices at Airbus. We report on lessons learned when applying this approach to Program Development Plans (PDP) based on business process models. Based on our experience, we also discuss open problems of interest for model-based and SPL engineering.

Remainder. Section II discusses the motivation of this paper; Section III introduces *Greal* using the EPL example; Section IV sketches the *Greal* semantics, i.e. our derivation algorithm; Section V reports on our experience of applying this approach at Airbus on Program Development Plans based on BPMN; Section VI discusses lessons learned, limitations and future work; Section VII describes research works related to the topic of this paper, and finally Section VIII concludes.

II. Background and Motivation

Two different kinds of techniques can be used to introduce variability into programming/modeling languages [9]: amalgamated and separated (aka orthogonal). The amalgamated approach proposes to augment the base language with variability concepts. In [7], [10], [11], the authors extend the UML metamodel for modeling variability in multiple UML diagrams, *e.g.,* Class or Sequence diagrams. In [12], [13], the authors propose a generic solution that can be applied to any kind of language and is fully supported by a tool.

In a separated approach (aka. orthogonal approach), the base language and the variability language are kept distinct and are related via a *mapping*. An illustration of this second approach

can be found in [14] (supported by the FeatureMapper tool) or in [15]. These approaches directly relate features and model elements and derive product models by removing all the model elements associated with non-selected features. Another example of the separated approach is VML* [16], which proposes a family of textual languages dedicated to the modeling of relationships between elements. VML* and FeatureMapper have been compared in [17] in terms of automation, scalability, expressiveness and evolution of the mapping.

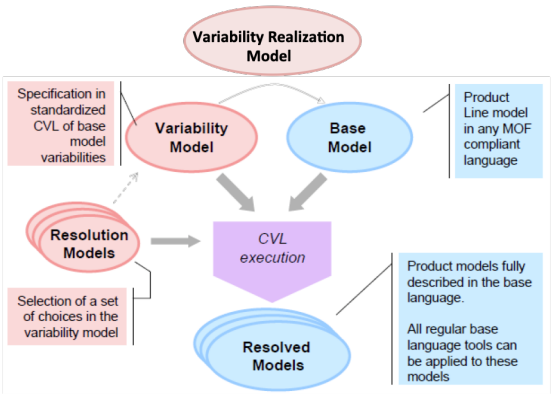

Fig. 1: Principle of Orthogonal Variability (from CVL)

In this paper we set ourselves in a context of an orthogonal approach where feature models are then linked to a base model through a *realization* model. The *Common Variability Language* (CVL) [18] has been unsuccessfully proposed as a standard for this kind of approaches, but it is nevertheless an excellent illustration of the important concepts that we need, as illustrated in Fig. 1.

CVL has been designed as a domain-independent language for specifying and resolving variability over any instance of MOF-compliant metamodels. The main concepts of CVL are:

- A *Variability Model* (VM), that provides a tree-based, high-level description of the SPL (domain space) in terms of features/decisions and their constraints, inspired by feature models [19].
- *Base Models* (BMs), i.e., a set of models, each conforming to a domain-specific modeling language (*e.g.,* UML). In CVL, a base model plays the role of an asset in the classical sense of SPL engineering. These models are then customized to derive a complete product.
- A *Variability Realization Model* (VRM), that specifies a mapping between features from the VM and model elements from the BMs. An SPL designer defines in the VRM what elements of the base models are removed, added, substituted, or modified (or a combination of these operations) given a selection or a deselection of a feature in the VM.
- *Resolution Models* (RMs), that store the choices made for a given configuration of the SPL. Realizations of the chosen features are then applied to the models to derive the final product model.

The VRM specifies whether a model element, or a set of model elements, change during the derivation process: removal (aka. *negative variability*), addition (aka *positive variability* [20]) or value assignment (in place modification of some model attribute or link).

As already noticed in several studies (see [21] for instance), CVL suffers from 3 main issues:

1) VRM operators work at a very low level, in terms of instances of a metametamodel (*e.g.,* MOF), *i.e.,* objects and links representing, *e.g.,* classes/attributes/methods for a class model, or states/transitions for a Statechart.
2) VRM operators are imperative: their order matters, which makes it difficult to reason about them or even just compose them (no associativity or commutative properties).
3) Since base models are expressed in their own specific language (*e.g.,* UML, SysML, BPMN, etc.), there is a need to specialize the VRM operators. Stated differently, removing a class in a class diagram or removing a transition in a transition system has a specific meaning, and needs a specific syntax and semantics.

In this paper we address these issues by proposing a new modeling language called *Greal* (standing for Generic Realization Language) to describe realization models in the spirit of CVL's VRM, but with the following key features:

- *Greal* is a generic high level language, where the concepts of the metamodels of interest can still be manipulated directly, *e.g.,* in terms of classes/attributes/methods for a class model, or states/transitions for a Statechart. For that, all the metaclasses of a given meta-model are imported in a *Greal* model. They can then be used as functions returning all the model elements of their type that match a given pattern. For instance, with UML, class(name='Exp.*') would return all classes from a base model whose names match the regular expression 'Exp.*'.
- *Greal* is a declarative language, to allow composition and reasoning over a realization model. In particular the *Greal* interpreter is in charge of computing the order of realization operations among the implicit partial order given in a *Greal* model. Even though *Greal* is declarative, it still supports both positive and negative variability, by either *extending* existing base models or *slicing* them. Composition of realization operations is possible, since negative variability is considered after all positive variability operations have been executed, and is idempotent.

III. Approach

In this section, we explain the approach using the *Expression Problem Product Line (EPL)* as described in [22] as a running example. We start with the UML models (M1) for the EPL and the corresponding Feature Model. We then show a realization model (M1) in *Greal*, and apply it to derive a product from the EPL. We finally present how this realization language was obtained from our generic metamodel.

A. The Expression Problem Product Line

As in [22] we use the following grammar for expressions:

Exp ::= Lit | Add | Neg Lit ::= non-negative integers
Add ::= Exp "+" Exp
Neg ::= "-" Exp

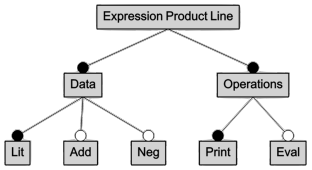

Fig. 2: The Feature Model for the EPL

Two different operations can be performed on the expressions described by this grammar: *printing*, which returns the expression as a string, and *evaluation*, which computes the value of the expression. The set of products in the EPL can be described with a feature model, as illustrated in Fig. 2. It has two orthogonal feature sets, the one concerned with the desired expressiveness of expressions, namely *Lit*, *Add*, and *Neg*, and the one concerned with operations that can be performed on expressions, namely *Print* and *Eval*.

Although simple, the feature model is representative of the kind of modular decomposition problems we find in the real world. In this example, initially *Lit* and *Print* are mandatory features. The features *Add*, *Neg* and *Eval* are optional. Feature models also support OR and XOR feature groups, as well as *includes* and *excludes* cross-tree constraints between features. Feature models have even been extended to support more complex constraints between features [23]–[25]. Fortunately, the complexity of inter-feature constraints makes no difference for our approach. *Greal* just cares about which features are selected, and which ones are not, and based on that, determines which models to compose or slice.

B. Base UML Models (M1) for the EPL

The EPL is implemented as 6 distinct models (each being valid w.r.t. the UML metamodel), carefully designed to work together (see Fig. 3) in the context of positive variability.

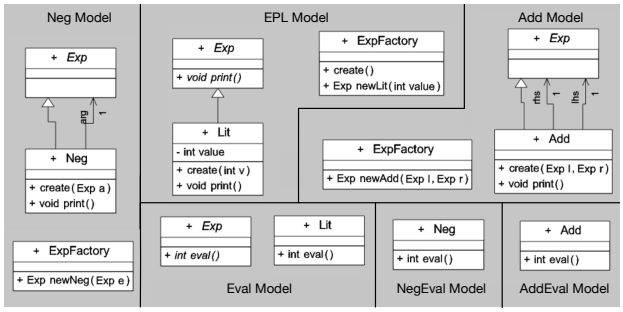

Fig. 3: Base UML Models for the EPL

C. Realization Model (M1)

To also illustrate negative variability, we assume that the modeler now changed their mind and wants the feature *Print* to be optional, so that when it is not selected, the corresponding model elements should be removed from the resulting product.

To describe how products can be obtained from the base models of Fig. 3 and a specific resolved feature model, we need a realization model, as presented in Fig. 4.

```
1  import FM "EPL-FM.xml" // Import Feature Model of Fig. 2
2  import UML "EPLModel.uml" as eplModel // Import EPL Model of
3                                        // Fig. 3 as a UML Model
4  Selected Add then { // Add is from the FM
5    import UML "AddModel.uml" as addModel
6    with addModel.Class(name="Exp") => eplModel.Class(name='Exp')
7    // since Greal supports automatic matching of model
8    // elements with identical names, this can be omitted
9  }
10 Selected Eval then {
11   import UML "EvalModel.uml" as evalModel
12   with evalModel.Class('Exp') => eplModel.Class('Exp')
13   // Class("Exp") is a shortcut for Class(name='Exp')
14 }
15 Selected Add, Eval then {
16   import UML "AddEvalModel.xmi"
17 }
18 NotSelected Print then {
19   remove eplModel.Method("print") // regular expressions
20 }
```

Fig. 4: A Greal Realization Model for the EPL

Our declarative realization language *Greal* makes it possible to (1) import a feature model (as in line 1), which renders the names of all its features visible; (2) import a base model conforming to a given metamodel, e.g., UML (as in line 2); and (3) specify that if a feature is *Selected* (e.g., *Add* in line 4) or *NotSelected* (e.g., *Print* in line 18), then one or several of the following operations must be performed:

- import another model and merge it with the base one (the imported model conforming to the base metamodel).
- *unify* existing model elements with newly imported ones as specified with the "=>" operator. To save typing and speedup most common cases, model elements of the same metatypes and the same names are automatically unified. In fact, all model elements from the newly imported model are treated as formal arguments that can be given an actual value from the current model (hence yielding a unification or join point), or from another source, *e.g.,* to transfer feature attributes to model elements or to rename an imported model element to avoid a name clash.
- remove specific model elements (last line of Fig. 4).

D. Resolved Model (M1)

For instance, if we would select the features *Add* and *Eval* (and not *Neg*), *Greal* would compose *EPL Model* with *Add Model*, *Eval Model* and *AddEval Model* and slice the result to remove the `print` methods. How this is done concretely is described in more detail in subsection IV. The resulting model is presented in Fig. 5.

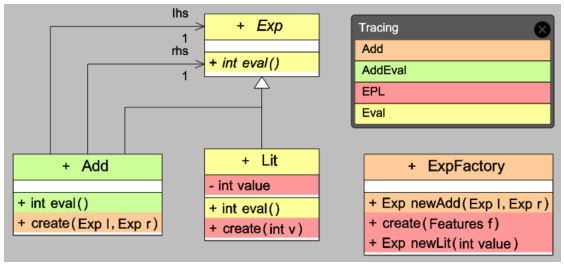

Fig. 5: EPL Model for Selection +Add,+Eval,-Neg, -Print

E. Instantiating Greal infrastructure for a Given Domain Specific Modeling Language

Since a realization *model* can refer to specific model elements through their type, the realization *language* infrastructure (static checker, lexer, completion engine) needs to be customized for each base metamodel to offer all of its metaclasses as type-checkable keywords.

In *Greal* this is achieved with the notion of *Perspective*, which makes it possible to import all the metaclasses of the corresponding metamodel (*e.g.,* UML) as *function* identifiers, *e.g.,* Class() as in the example of Fig. 4. These functions return model elements of their corresponding type using pattern matching on the metaclass attribute values. All attributes of a metaclass ('name' or 'isAbstract') turn into accepted named parameters of the function. For instance, class(name='Exp.*') would return all classes from a base model whose names match the regular expression 'Exp.*'. Since a perspective always conforms to exactly one well-defined metamodel, static type checking of a *Greal* model is straightforward, with all the expected benefits in terms of early error detection, but also advanced editing support.

Fig. 6 summarizes the main concepts of *Greal* in the form of a simplified metamodel. This excerpt shows the three main parts of the metamodel.

- The algebraic language for expressing conditions on the selected features that trigger actions (grey part),
- the action language allowing to define a set of actions among which the slice (*RemoveAction*), the value assignment (*SetAction*) and the import of another model to merge it (*PositiveImportAction*) (red part).
- the *RealizationModel* that references a *FeatureModel* and a *Model* conforming to a *perspective*. The *RealizationModel* has a set of steps that can specify a set of rules to bind features algebraic expression and a set of *Actions*.

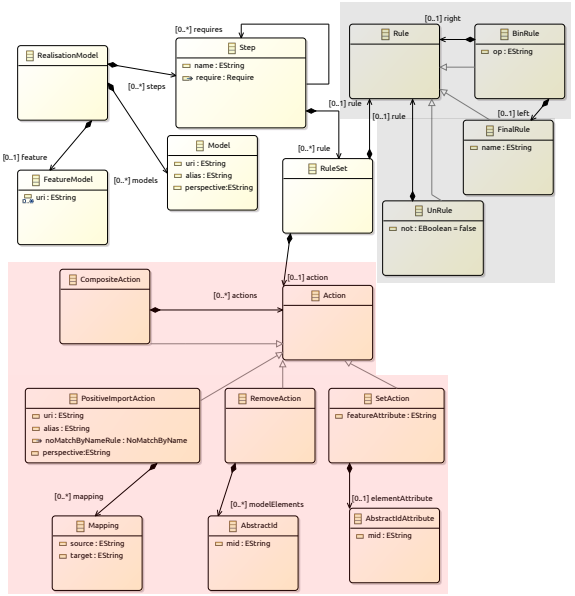

Fig. 6: Greal Simplified Metamodel

IV. Greal Semantics: the Derivation Algorithm

The specification of a realization model is declarative to allow composition and reasoning. Still some sequencing is often useful. This leads to a multi-steps specification, where each step is a set of declarative realization operations (existence, substitution and value assignment).

A. One-step Derivation Algorithm

Given a valid feature selection and a realization model, the derivation algorithm of a given step of the realization model derives a (possibly intermediary) model by first resolving the positive variability (phase 1), then addressing negative variability (phase 2), and finally value assignments (phase 3). An additional phase (phase 4) is also conducted to normalize the resulting model, both in terms of the base language and specific modeling practices from the domain.

Guarded by whether any given set of features are **Selected** or **NotSelected**, each clause from a realization model can import any number of model elements, optionally rename them and unify them with base model elements (positive variability), remove any model element (negative variability), and can assign values to model elements, those values coming either from the realization model itself or from feature attributes. Since features can appear any number of times in a realization model, their associated clauses form a directed acyclic graph (DAG). For instance in the realization model of Fig. 4, the clause of line 15 where *Eval* and *Add* are both selected is *below* the clause of line 10 (Eval) and line 5 (Add). This allows feature interactions to be dealt with, as explained below.

The derivation algorithm, which is inspired from the CORE derivation algorithm for positive variability presented in [26], first determines the set of clauses *Rcons* that are to be considered as follows. Given a valid feature model configuration C containing selected and unselected features, the algorithm first removes all features from C that do not have any associated clauses. The algorithm then looks for a clause that realizes (or realizesNot) all the remaining features in C (i.e. for which the features are respectively **Selected** or **NotSelected**). If there is no such clause, the algorithm looks for clauses that realize (or realizeNot) a subset of features in C of size —C—-1, then of size —C—-2 and so on. Once one or several clauses are found, they are added to the list of clauses to be considered in *Rcons*, and the features they realize are removed from C. The algorithm continues until C is empty.

This way of determining which clauses to use makes it possible to elegantly handle feature interactions. Feature interactions describe situations where the realization of a product has to be adapted because of the *simultaneous presence (or absence)* of several features. For example, in the EPL, the *AddEval* model is a model that deals with the feature interaction of *Add* and *Eval*. Only if both *Add* **and** *Eval* are selected, the class `Add` needs an `int eval()` operation.

Finally, starting with the models in *Rcons*, the algorithm then follows all dependencies to determine the set of realization operators *Rprocess* that are going to be processed.

1) Resolving Positive Variability: To resolve positive variability, the derivation algorithm first traverses the DAG in top-down order. Whenever an import is encountered, the algorithm invokes the model composition algorithm of the realization (modeling) language, passing unification mappings as a parameter, if any. By default, model elements are unified through explicit mappings. Possibly, model elements with the same name or signature are automatically unified. The algorithm then takes the resulting model and continues the DAG traversal. To this aim, the remaining imports in the DAG are updated to point to the model resulting from the previous composition, and then the next import is handled. Removals are ignored during the DAG traversal of phase 1.

Whenever a model A is composed with a model B, any unification mappings that map model elements from B to A that were applied during the composition of the models must also be applied to follow-up compositions of phase 1. For example, if there are unification mappings from a model C to model B, then the unification mappings from C to B must be transformed into C to A mappings using the B to A mappings.

2) Resolving Negative Variability: The negative derivation is based on the primitive remove operator within the realization language to remove a set of model elements. Again, the derivation algorithm traverses the DAG that by now only contains removals and value assignments in a top-down order. For each clause, the corresponding directives are processed as follows:

1) We compute the set of model elements Me_remove explicitly mentioned in the remove declarations. Remove statements can reference a set of model elements based on the metaclass name, pattern matching on the name attribute, if it exists, or based on model element id.
2) We recursively compute the model elements that are related to this first set of model elements that were designated by the remove statements, because they also need to be removed, and update Me_remove. The related model elements could be: i) contained model elements (based on the metamodel structural constraints). For example, in UML, designating a method to be removed would also remove the method's parameters ; ii) model elements referencing the elements that are to be removed (by default, if the deletion of the element leads to the violation of the minimal cardinality of the relation linking these two elements, then the linked element is also deleted).
3) We update relations in the model referencing model elements contained in Me_remove to remove references.
4) We delete all elements contained in Me_remove from the model. We use the delete mechanism that provides a specific semantic for the selected metamodel if it exists (i.e., the base language). If it does not exist, we use the generic way to remove a bulk of model elements from a model. In case the deletion of an already deleted model element is requested, no error is signaled to the user.

3) Resolving Value Assignments: Finally, the derivation algorithm applies the value assignments, in the depth-first order of the DAG. However, such a sequence of value assignments is generally not commutative nor associative if several assignments target the same attribute. There are 3 possible ways to handle this issue. The most restrictive one is to allow the assignment of a given attribute only once. A second possibility would be to interpret any assignment as a relative assignment, i.e. $x+ = 3$ or $x* = 3$ with a static check ensuring that only one kind of operator is used all along for each attribute to preserve associativity and commutativity. The third possibility would be to partially give up the declarative aspect of *Greal* and rely on the order given by the designer. At the request of Airbus, this choice is implemented as a semantic variation point that can be configured by the *Greal* user.

4) Model Normalization: Once the model has been derived from the two aforementioned phases, a last phase of normalization is applied to both make sure the resulting model conforms to the base language (if not initially ensured by the composition and removal operators), and to apply specific modeling practices. This phase consists only in refactoring the model, without any semantic change of the model. Hence, the input and output models have the same semantics, i.e., both models simulate each other.

B. Realization Model Composition and Multi-step Derivation

As explained above, we introduce a declarative formalism to specify the realization model. To ensure a sound and valid derivation of any model, we first apply positive variability management (i.e., composition) and then negative variability management (i.e., removal).

Composition is done in two steps: first the import of the new model, with possible renaming and attribute setting, and then the merge of this imported model with the current one, using unification of model elements of the same type and the same names.

The semantics of removal is idempotent: if a remove operator is called twice on the same element, the result is that the element is removed. Therefore, to ensure the declarative nature of our formalism and the validity of the derivation process, we first apply positive variability management and then negative variability management. Value assignments applied in phase 3 of the one-step derivation algorithm must be restricted as discussed before.

Depending on the domain and the base language, some situations are more convenient to address with an interleaving of positive and negative realization operators. Such complex workflows would require applying negative variability after the application of positive variability and value assignments. To keep the derivation of any model according to a given configuration valid, the execution flow has to be made explicit, with an intermediate and propagated derived model in between a negative realization operator and a following positive realization operator. The formalism would be no longer declarative, but a succession of declarative steps that can be seen as a multi-step derivation algorithm. The specification of the workflow (i.e., sequence of steps) can be either expressed internally in the realization model or externally through a dedicated workflow language.

In practice, the proposed realization language supports the definition of a set of steps, all possibly using the resulting models of previous steps. This creates a partial order to process the steps, possibly even enabling parallelization.

V. Experimentation at Airbus

A. Program Development Plans at Airbus

A Program Development Plan (PDP) is a description of a highly integrated development thread from program preparation to handover to series. It highlights the key deliverables, decisions and the maturity levels to be achieved at each step of a program. It helps to identify the contribution of each stakeholder in a long and complex ecosystem. It inherits a long history of aircraft project management experience and incorporates a large amount of lessons learned (e.g. A380, A350, NEO...).

PDPs are part of an Airbus-wide digital transformation program, publicly known as DDMS. Digital Design, Manufacturing & Services (DDMS) is a digital-first approach to the way aerospace products are designed, manufactured, and operated. At Airbus, the DDMS program focuses on digital methods and tools for end-to-end business processes that ensure we can reduce costs and time to market for our products, while meeting our customers' expectations for quality, safety and environmental performance.

During the early phases of a program, the uncertainty on the future program assumptions is still high (e.g. mission, range, make or buy policy, number of aero load stress loops), and consequently the associated PDP is not stable yet. In the context of DDMS and to improve cost evaluation and time-to-market calculation efficiency in these early phases, the Airbus PDP Architects decided to explore variability management on end-to-end business process simulation using a standard business process modeling language named BPMN (Business Process Model and Notation). PDP Architects select BPMN as it is formal enough to be simulated (in addition with the BPSim extension for lead time and resources calculation) and simple enough to be directly used by businesses.

B. Implementation

To perform the evaluation of our approach, we developed both a generic implementation of *Greal* and set of specialisations for BPMN. The implementation is based on Eclipse (https://www.eclipse.org/) and Xtext (https://www.eclipse.org/Xtext/) for the definition of the realization language. For the context of the case study built on top of BPMN, we use the Eclipse BPMN editor (https://www.eclipse.org/bpmn2-modeler/) and the ELK project (https://www.eclipse.org/elk/). The motivation for these choices was mainly driven by the possibility to have access to the source code of these tools. Feature models are defined using the open-source tool FeatureIDE [27] (https://featureide.github.io/). Feature IDE provides an attributed feature model editor with support for constraints based on boolean expressions. It also provides a simple configuration editor with support for constraint propagation. It is therefore possible to select the features we want for a particular model instance. Currently, it is possible to select or not an option, to make a partial selection, and to assign values to some attributes defined in the feature model.

Our implementation provides a pluggable architecture to extend the generic behavior of Greal for a given domain-specific modeling language (e.g. BPMN). Two kinds of extensions can be provided for a specific modeling language:

- specific remove semantics for model elements that bypass the containment property that exists at the metameta model level (e.g. removing a gateway means, e.g., removing the associated diagram element and BPSIM element).
- normalization rules to automatically refactor the BPMN model at the end of each step. For implementing a normalization rule, current implementation consists in providing a Java class that implements a specific interface. Each developer of a plugin for a specific normalisation must guarantee that the transformation is a refactoring[1].

C. Generic Building Blocks for the Domain Modeler

At Airbus, Domain Engineering starts with very basic BPMN building blocks, here named Generic Patterns (GP), as presented in Fig. 7: GP – Aero load stress loop (X) ; GP – Produce SAM (X) ; GP – Data drop (X).

D. Business Level Patterns

From the kind of building blocks described above, the Domain Modeler can assemble a library of Domain BPMN Components. For instance, with following Greal Realization Model, she can compose the 3 basic models of Fig. 7 to obtain a generic Manage Aero load stress loop as represented in Fig. 8:

```
Step 1:{
import BPMN "GP - Aero load stress loop (X).bpmn" as StressLoop
remove StressLoop.EndEvent("*")
import BPMN "GP - Produce SAM (X).bpmn" as Sam
with Sam.Task("Produce SAM (Y)").input =>
    StressLoop.IntermediateThrowEvent(
        "FeasibilityStudy (X)").output
remove Sam.IntermediateCatchEvent("FeasibilityStudy (X)")

import BPMN "GP - Data drop (X).bpmn" as DataDrop
  with DataDrop.Task("Produce and provide data drop (X)").input =>
    Sam.IntermediateThrowEvent("SAM (Y)").output
remove DataDrop.IntermediateCatchEvent("SAM (Y)")

forall Sam.task("Produce SAM (Y)")
    set leadTime = FixDuration("P264M")
}
```

E. Business Components Ready for Use

Generic Business patterns as the Manage Aero Load stress loop (X) of Fig. 8 need to be instantiated as ready to use Business Components to simplify the task of the Application Modeler. For instance, in a typical Airbus PDP, the Aero Load stress loop has to be executed 3 times with increasing levels of maturity, yielding models called Loop1, Loop2 and Loop3. In some cases, there might even need an additional loop, called Loop2bis. All these variants of the Manage Aero Load stress loops are obtained by instantiating the meta-parameters X, Y

[1]Restructuring of a model to improve its internal qualities without changing its external behavior.

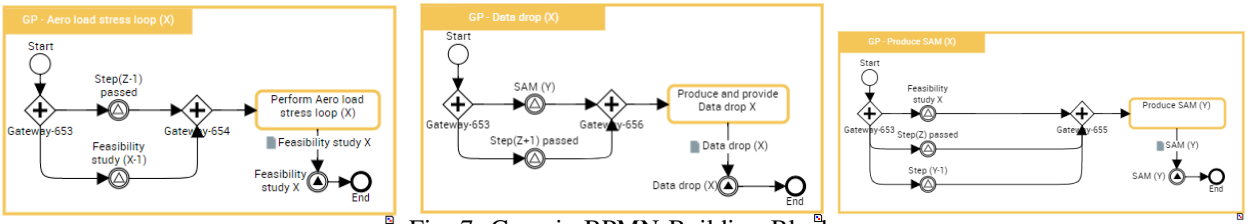

Fig. 7: Generic BPMN Building Blocks

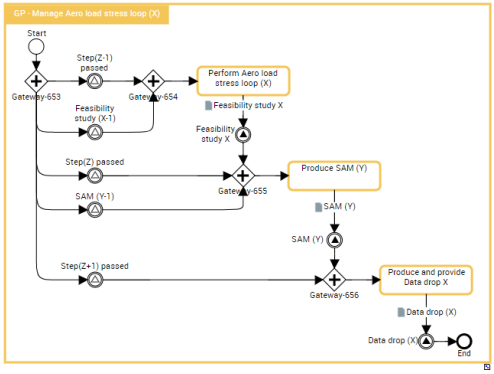

Fig. 8: Manage Aero load stress loop

and Z to actual values, such as $X-1 = Decision$ for Loop1, as illustrated in Fig. 9. Since the various Aero Load stress loops feature different lead times, the realization model also has to update these with relevant values, either in an absolute way or relative to the value already existing in the base model.

As an example, here is an extract of the code for obtaining the Loop1 model.

```
Loop1 require Step1 as base: {
forall base.BaseElement(".*X-1.*")
    set name = "\(1)Decision\(2)"
forall base.BaseElement(".*X.*")
    set name = "\(1)1\(2)"
...
forall base.Gateway("*") set name = "\(1)-1";
remove base.IntermediateCatchEvent("SAM (Y-1)");
forall base.Task("Perform Aero load stress loop (X)")
    set LeadTime = 'FixDuration("P132M")' ;
}
```

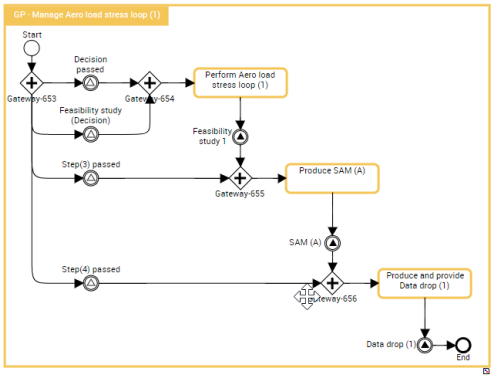

Fig. 9: Manage Aero load stress loop 1

F. Variability Model

The variability model capturing the various features of the Airbus PDP contains several hundreds of features with many cross-tree constraints. For confidentiality reasons, we can only discuss an extremely simplified variability model for the PDP, related to the manner in which its Aero Load stress loops are articulated. Depending on the level of novelty of an aircraft building process, the PDP can be configured by selecting one of the following three features.

- Nominal: 3 Aero Load stress loops are performed: Loop 1, Loop 2 and Loop 3.
- Conservative: 3 Aero Load stress loops are performed: Loop 1 Simplified, Loop 2 and Loop 3. Also, some lead times should be adjusted to model the fact that some processes are well known and feature less variability.
- Innovative: 4 Aero Load stress loops are performed: Loop 1, Loop 2, Loop 2bis and Loop 3. Here some lead times will also be modified, but in the opposite direction since the processes are supposed to be less understood in this case.

The following realization model makes the connection between the variability model and the BPMN models presented in the previous section, including the modification of lead times.

```
import FM "model.xml";
import BPMN "Loop1.bpmn" as Loop1;
ManageAeroloadStressLoop: {
Selected Nominal then {
import "Loop2.bpmn" as Loop2 noImplicitMatchByName
with Loop2.Task("Produce and provide Data drop X") =>
    Loop1.Task("Produce and provide Data drop X")
with Loop2.IntermediateCatchEvent("Step3 passed") =>
    Loop1.IntermediateCatchEvent("Step3 passed")
with Loop2.IntermediateCatchEvent("Step4 passed") =>
    Loop1.IntermediateCatchEvent("Step4 passed")
with Loop2.StartEvent("*") => Loop1.StartEvent("*")
with Loop2.EndEvent("*") => Loop1.EndEvent("*")
with Loop2.Gateway("655-2") => Loop1.Gateway("654-1");

import "Loop3.bpmn" as Loop3 noImplicitMatchByName
with Loop3.Task("Produce and provide Data drop X") =>
    Loop1.Task("Produce and provide Data drop X")
with Loop3.IntermediateCatchEvent("Step4 passed") =>
    Loop1.IntermediateCatchEvent("Step4 passed")
with Loop3.IntermediateCatchEvent("Step5 passed") =>
    Loop2.IntermediateCatchEvent("Step5 passed")
with Loop3.StartEvent("*") => Loop1.StartEvent("*")
with Loop3.EndEvent("*") => Loop1.EndEvent("*")
}

Selected Conservative then {...}
Selected Innovative then {...}
}
```

For instance if the feature "Nominal" is selected, the resulting BPMN is as shown in Fig. 10.

G. Normalization Rule

In the context of BPMN & BPSIM standards applied to the engineering of PDPs, we apply the following normalization rules: i) Removal of duplicate sequence flows (i.e. same source and target model element) resulting from the composition of

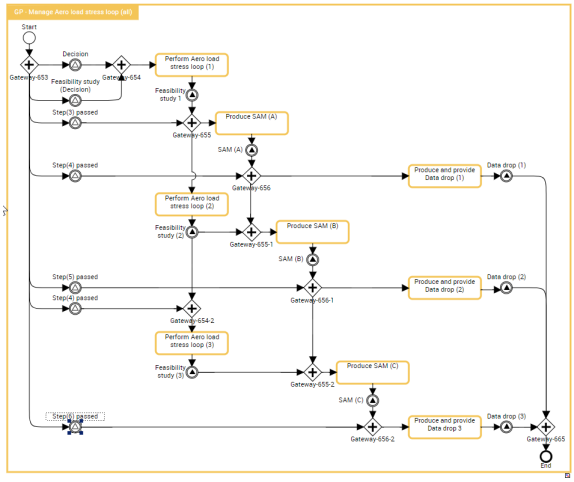

Fig. 10: Manage Aero load stress loops

input models; ii) When a sequence flow needs to be bound to a target model element, this one is removed if both the source and target are gateways; iii) Removal of useless gateways, i.e., with only 1 input, resp. 1 output; iv) Removal of events (send/receive) when an explicit sequence flow exists.

VI. Lessons learned and Open Challenges

We now discuss different lessons learned of applying *Greal* at Airbus, some leading to further open challenges of interest for the MDE and SPLE community.

A. *Generic* Vs. *Specific Realization Language*

In this paper, we introduce the first generic high-level realization language that supports the seamless and sound combination of positive and negative variability management. Using *Greal*, we have been able to deliver a realization language that fits the specifics of BPMN and modelling practices at Airbus. To do so, we benefited from the *Greal* generic metamodel (Fig. 6) but we also had to specialize different parts. In fact, there is a natural tension between genericity and specificity. Specifically, the genericity comes from the fact that the realization language is built in such a way that it can be applied to any base model defined with an Ecore-based language, taking into account the native structural constraints supported in the specification of an Ecore metamodel. Hence, the realization language is agnostic to the base language, but also to the specific semantics coming from the base language (and not captured as structural constraints in the language metamodel). For instance, the removal of a model element can lead to an arbitrarily complex behavior according to the base language semantics (removing a class in a class diagram could lead to the removal of the associated references, while removing a state in a statechart could lead to reconnecting the transitions to the following state). While CVL is generic, it achieves this genericity by working at a low level of abstraction, where models are treated simply as objects with references. As a result, a direct usage of CVL would not take into account any structural or other constraints specified in the language's metamodel. In response, we offer the ability to specialize semantic constraints through the capitalization in a library associated to the base language, and then plugged into the *Greal* semantics. To specialize the realization language, we also introduce a phase dedicated to model normalization. While this phase is limited to the refactoring of the resolved model, hence not changing the overall model semantics, it makes it possible to take into account modeling rules specific to the language, or even to a particular domain or use. In our prototype implementation, these rules are easily plugged on demand, and automatically applied after each derivation of a complete step. This ensures that intermediate models conform to the domain-specific modeling rules.

Development effort. Overall, the development effort for specializing the realization language was mainly focused on 1) a specific removal operator with the challenge of correctly propagating the removal of the associated elements in a way that respects the syntax and semantics of BPMN as well as internal modelling rules at Airbus; 2) a set of refactoring rules to normalize and improve the output of the derivation. This effort required a lot of back and forth and exchanges with various Airbus stakeholders to refine the tooling according to the practices at Airbus. The current implementation contains 45652 lines of Java code for the generic part (incl. 42088 generated Xtext code, using *cloc* to compute the number of lines of code) and 903 for the specialization for BPMN. This illustrates the reuse provided by *Greal*.

B. *Combining Positive and Negative Variability*

Positive and negative variability management are two different approaches with pros and cons concerning expressivity. Negative variability management is limited because the base 150% model must conform to the base language; (e.g., a class attribute cannot have two types), and the inability to specify unlimited reuse of specific model patterns (e.g., application of a generic pattern in a particular context). Conversely, positive variability management can lead to the multiplication of small model patterns to support fine-grained configuration of the base model. Positive variability management can also lead to complex composition operators to ensure a correct assembling of the model patterns, not only specific to the base model language but also to the modeling practices. Both mechanisms have demonstrated to be necessary and therefore should be useable in combination.

When to use positive or negative variability? Hence, a key open question is *when* modelers should rely on positive and/or negative. From the authors' experience, there is contextual information to consider from the application domain when choosing one or the other approach. A general answer is unlikely, and one rather needs a specific method for a particular engineering context (e.g., the one defined by Airbus for PDP engineering). At Airbus, we observed that positive variability management allows the designer to choose only the features that interest her without requiring a global view of all the possible configurations. A global view might be beyond the scope of the current configuration under definition, and thus beyond the designer's expertise at this time. Positive variability also

allows modellers to avoid repetition of processes. Conversely, negative variability management requires a designer to face all possible variants, and to explicitly indicate how to slice the base model based on a given configuration. While this requires the designer to understand all choices well, it also more easily supports trade-off analysis between the different possible alternatives. In practice, at Airbus we observed the need to combine the two approaches according to the context for a given part of the configuration.

Tradeoff in the realization model. Relying on positive variability, the domain engineering phase has the advantage of making it possible to work with the real mental building blocks that a PDP architect has in mind, closed to a declarative way. Leveraging *Greal*, reusable model fragments can be capitalized and composed for exploring different PDP alternatives. The obvious drawback is that while the base models are much simpler, the realization models are becoming much more complex. The knowledge of how to compose basic building blocks is now concentrated in realization models, moving the complexity into the realization as opposed to expressing it in the base model. It is interesting to notice that the opposite effect is observed with negative variability, where the complexity resides in the base. We consider that part of this complexity is accidental, and could be removed with even more abstract process descriptions. In the PDP, there are indeed many cases of generic processes that produce related documents or events. For instance, a process might consume a "Feasibility Study (X-1)" and produce a "Feasibility Study (X)". Instead of treating these as uninterpreted strings, which makes the substitution quite heavyweight during instantiation, we might actually consider them as array variables. A syntax usually found in programming languages could be adopted, i.e. using brackets [], with an index variable called i. For instance: "FeasibilityStudy[i]" or "FeasibilityStudy[i-1]". Then the array "FeasibilityStudy[]" could store the wanted values of the event/document for each possible value of i. Hence, more idiomatic constructs can emerge to ease the tasks of modellers. In general, the concrete syntax and integrated tooling that comes with the realization language deserves further investigation (see also Section VI-D).

C. Positive and Negative Variability Management

Valid combination of positive and negative variability. In our approach, we introduce a declarative formalism to specify the realization model. However, to ensure a valid derivation of any model, we first apply positive variability management, then negative variability management, and finally value assignments. Composition is done in two steps: first the *import* of the new model into the work space, with possible renaming and attribute setting, and then the *merge* of this imported model with the current one, using unification of model elements of the same type and the same names. The semantics of removal is idempotent: if a *remove* operator is called twice on the same element, the result is that the element is removed. So the order of removal does not matter, and can thus safely be applied after all other operations.

According to the domain and the base language, it could be more convenient to have an arbitrary combination of positive and negative variability management. To keep the derivation valid for any model according to a given configuration, this will require to explicit the execution flow, with an intermediate and propagated variability model in between a negative variability management operation and a following positive variability management operation. The formalism would be no longer declarative, but a succession of declarative steps that could be seen as a multi-staged configuration [28]. Hence, we found a pragmatic tradeoff between declarative and imperative-style instructions in the realization language.

Towards a method for managing variability at Airbus. The *Greal* language and the prototype implementation introduced in this paper must be complemented with a dedicated method to identify the various activities related to domain and application engineering. To provide a comprehensive view, it is envisioned to explicitly model the workflow of the various steps involved. This approach allows the steps to be applied sequentially or in parallel, in different orders, resulting in a more intricate flow. Additionally, each step could be assigned to different stakeholders, supporting different governance, roles and associated expertise, which can also be specified. Moreover, the variability management method must ensure the coupling with the use of a versioning system for variability in time, and the support of collaborative modeling techniques.

D. Flexible Syntax of the Realization Language

The need of flexibility for the syntax. We can distinguish three phases in the lifecycle of a realization model: the initial writing of the specification; the reading and understanding (possibly by a stakeholder different from the one who has written the specification: domain experts or other modelers); the maintenance and evolution that requires both an understanding and a writing of the specification. In all cases, the visualization of the variability, being written with negative, positive, or a combination therefore, is important. Ideally, BPMN models should keep the intention of the domain and experts; with variability information, there is a risk to alter the information and overload modelers or experts. Hence, the question on how to present variability within the BPMN models arises. For instance, some colors can be used to trace features to model elements in the 150% model and in the case of negative variability. In the case of positive variability, available and reusable model fragments that can be assembled can be depicted, but there is then the challenge of visualizing where the fragments are composed. Grouping together "closed" model fragments that are subject to composition can ease their comprehension and reuse. Hence, the syntax of the realization model should be flexible and tailored to the modeling activities.

Towards metamorphic realization language. Right now, the realization models can be written using a textual, external DSL – as traditionally done for specifications. However, it should not be the only implementation choice. First, there can well be internal DSLs, for example Java fluent APIs. The advantage of an internal DSL is a smooth integration to the

host language (e.g., Java) and a possible reuse of mainstream tooling. However, an internal DSL might be more verbose and disconnected from a specific ecosystem (e.g., the BPMN tooling). A possibility is thus to offer internal DSLs, hosted on top of Java, Python, etc., in addition to the external DSL. A "must" would be to transition from external to internal DSL in an automated way: a realization model written in the external DSL could well be edited through a fluent API (and back again). There are software language engineering works for supporting the construction of several "shapes" of a language [29], and there have been some attempts to provide different shapes of a feature modeling language [30]. Second, the use of another "shape" of the realization language can be envisioned into spreadsheet tools like Excel. The idea would be to offer Airbus modelers the ability to specify the mapping between individual features (or combinations of features) and model elements or fragments with a spreadsheet. This proposal has to be carefully assessed. The intended benefits are to improve the usability of the language with a tabular-like specification and an integration into well-known tools.

VII. Related Work

The Common Variability Language (CVL) has emerged to provide a solution for managing variability in any domain-specific modeling languages [18]. The effort involves academic and industry partners and pursues the goal of providing a generic yet extensible solution. Our technical contribution can be seen as a methodology to implement CVL and instantiated to fit the modelling needs at Airbus. We also report on experiences and lessons learned when specializing CVL in such a context. CVL exhibits similarities with several variability approaches that we now discuss.

Voelter and Groher et al. called to support the combination of negative and positive variability in various kinds of artefacts [20]. Variability implemented with compositional approaches is called positive variability, since variable elements are added together. Negative variability mainly refers to annotative approaches. Different approaches to represent negative and positive variability have been proposed. For instance, the directives of the C preprocessor (#if, #else, #elif, etc.) conditionally include parts of files (e.g., C or C++ code [31], [32]), and as such, can be used to model negative variability. Many techniques to realize compositional approaches can be found in the literature: frameworks, mixin layers, aspects [33], stepwise refinement [34], etc. In model-based SPL engineering, the idea is that multiple models or fragments, each corresponding to a feature, are composed to obtain an integrated model from a feature model configuration. Aspect-oriented modeling techniques have been applied in the context of SPL engineering [35], [36]. Apel et al. propose to revisit superimposition technique and analyze its feasibility as a model composition technique [37]. Superimposition is a generic composition mechanism to produce new variants – being programs (written in C, C++, C#, Haskell, Java, etc.), HTML pages, Makefiles, or UML models – through the merging of their corresponding substructures [38]. Perrouin et al. propose a flexible, tool-supported derivation process in which a product model is generated by merging UML class diagram fragments [39].

Delta modeling [40] promotes a modular approach to develop SPLs. The deltas are defined in separate models, and a core model is transformed into a new variant by applying a set of deltas. Delta modeling shares many similarities with CVL and similar specialization would be required to fit the requirements of the targeted modeling language and a large company like Airbus. Damiani et al. [41] present a core calculus that extends delta-oriented programming with the capability to switch the implemented product configuration at runtime. At the foundation level, the Choice Calculus [42] provides a theoretical framework for representing variations (being annotative or compositional). In [43], Filho et al. empirically investigated in the context of the Java language which CVL operators, applied to program elements, can synthesize variants of programs that are correct. A key result is that a large portion of variability transformations are unsafe and lead to incorrect derivation, hence requiring a significant effort to specialize CVL to Java. Our experiences at Airbus exhibit similar challenges when specializing CVL to the language (BPMN), but also to the modelling practices in a large organization. Some extensions have been proposed to deal with the diversity of modeling languages or to tailor CVL for specific engineering contexts [43]–[48]. To the best of our knowledge, however, very few works report on real-world experience of applying or specializing CVL.

There has been some empirical user studies about variability mechanisms [49]–[54]. It is unclear how these findings transfer to the Airbus context, owing to the usage of a different language (BPMN), modelling practices and targeted audience. As learned (see Section VI), methodologies and guidelines to combine both positive and negative variability are still missing, and most likely require to observe the specific needs of engineers and domain experts. We plan to conduct user studies with *Greal* at Airbus and also over different modeling formalism in other engineering contexts.

VIII. Conclusion

In this paper, we proposed *Greal* as a generic and high-level realization language that can be automatically composed with one or more base metamodels. We furthermore proposed a product derivation algorithm that applies a realization model to a base model and a resolved model to yield a derived product. Our algorithm makes it possible for the developer to deal with feature interactions, while supporting positive and negative variability. We illustrated our approach using the well-known use case of the Expression Product Line, highlighting the need for both positive and negative variability. *Greal* comes with a prototype open source implementation that has been tested with several base languages, including UML class diagrams, UML Sequence Diagrams, BPMN, Java (at the AST level), and also text files (seen as lists of lines) and Word documents (seen as lists of paragraphs, tables and figures). However we ran a large scale validation of Greal only on BPMN for the Airbus

PDP product line. Still it yielded some interesting feedback that are going to be useful for the future of *Greal* and model-based SPL engineering.

ACKNOWLEDGMENTS

The research in this paper was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2018-06610 as well as by the Junta de Andalucia, Spain, under contract QUAL21 010UMA.

REFERENCES

[1] L. Northrop, "A framework for software product line practice," in *Proceedings of the Workshop on Object-Oriented Technology*. Springer-Verlag London, UK, 1999, pp. 365–376.

[2] F. van der Linden, "Software Product Families in Europe: The Esaps & Cafe Projects," *IEEE Software*, vol. 19, pp. 41–49, 2002.

[3] G. Halmans and K. Pohl, "Communicating the variability of a software-product family to customers," *Software and System Modeling*, vol. 2, no. 1, pp. 15–36, 2003.

[4] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[5] F. Bachmann and L. Bass, "Managing variability in software architectures," *SIGSOFT Softw. Eng. Notes*, vol. 26, pp. 126–132, May 2001. [Online]. Available: http://doi.acm.org/10.1145/379377.375274

[6] N. Anquetil, B. Grammel, I. Galvão, J. Noppen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia, "Traceability for model driven, software product line engineering," in *ECMDA Traceability Workshop Proceedings*, vol. 12. SINTEF Norway, 2008, pp. 77–86.

[7] T. Ziadi and J.-M. Jézéquel, "Software product line engineering with the uml: Deriving products," in *Software Product Lines*. Springer, 2006, pp. 557–588.

[8] C. Nebut, Y. Le Traon, and J.-M. Jézéquel, *System Testing of Product Families: from Requirements to Test Cases*. Springer Verlag, 2006, pp. 447–478. [Online]. Available: http://www.irisa.fr/triskell/publis/2006/Nebut06b.pdf

[9] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," *Software Product Line Conference, International*, vol. 0, pp. 139–148, 2008.

[10] C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line development: the kobra approach," in *Proceedings of the first conference on Software Product Lines: Experience and Research Directions*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 289–309.

[11] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[12] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.-M. Jézéquel, "Weaving variability into domain metamodels," in *MoDELS*, 2009, pp. 690–705.

[13] G. Perrouin, G. Vanwormhoudt, P. Lahire, B. Morin, O. Barais, and J.-M. Jézéquel, "Weaving Variability into Domain Metamodels," *Software and Systems Modeling Special issue*, p. 22, 2010.

[14] F. Heidenreich, J. Kopcsek, and C. Wende, "FeatureMapper: Mapping Features to Models," in *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. New York, NY, USA: ACM, May 2008, pp. 943–944.

[15] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*, ser. LNCS, vol. 3676, 2005, pp. 422–437.

[16] S. Zschaler, P. Sanchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araujo, and U. Kulesza, "Vml* – a family of languages for variability management in software product lines," in *Software Language Engineering (SLE'09)*, ser. LNCS, vol. 5969. Springer, 2009, pp. 82–102.

[17] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. K. amd Ana Moreira, and A. Rashid, "Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*," *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, vol. 6210, pp. 69–114, 2010.

[18] Ø. Haugen, A. Wasowski, and K. Czarnecki, "Cvl: common variability language," in *Proceedings of the 16th International Software Product Line Conference-Volume 2*, 2012, pp. 266–267.

[19] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: a comparison of variability modeling approaches," in *Proceedings of VaMoS'12*. ACM, 2012, pp. 173–182. [Online]. Available: http://doi.acm.org/10.1145/2110147.2110167

[20] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *SPLC'07*. IEEE, 2007, pp. 233–242.

[21] J. B. F. Filho, O. Barais, M. Acher, B. Baudry, and J. L. Noir, "Generating counterexamples of model-based software product lines: an exploratory study," in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, T. Kishi, S. Jarzabek, and S. Gnesi, Eds. ACM, 2013, pp. 72–81. [Online]. Available: https://doi.org/10.1145/2491627.2491639

[22] R. E. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating support for features in advanced modularization technologies," in *European Conference on Object-Oriented Programming*. Springer, 2005, pp. 169–194.

[23] A. S. Karatas, H. Oguztüzün, and A. H. Dogru, "From extended feature models to constraint logic programming," *Sci. Comput. Program.*, vol. 78, no. 12, pp. 2295–2312, 2013. [Online]. Available: https://doi.org/10.1016/j.scico.2012.06.004

[24] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants," *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 1763–1794, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9462-4

[25] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France, "Composing your compositions of variability models," in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, Eds., vol. 8107. Springer, 2013, pp. 352–369. [Online]. Available: https://doi.org/10.1007/978-3-642-41533-3_22

[26] J. Kienzle, G. Mussbacher, P. Collet, and O. Alam, "Delaying decisions in variable concern hierarchies," in *International Conference on Generative Programming: Concepts and Experiences – GPCE 2016*. ACM, 2016, pp. 93–103.

[27] J. Meinicke, T. Thm, R. Schrter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[28] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," in *Software Product Lines: Third International Conference, SPLC 2004*, ser. Lecture Notes in Computer Science, vol. 3154. Heidelberg, Germany: Springer Berlin / Heidelberg, 2004, pp. 266–283.

[29] F. Coulon, T. Degueule, T. Van Der Storm, and B. Combemale, "Shape-diverse dsls: languages without borders (vision paper)," in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, 2018, pp. 215–219.

[30] M. Acher, B. Combemale, and P. Collet, "Metamorphic domain-specific languages: A journey into the shapes of a language," in *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, Eds. ACM, 2014, pp. 243–253. [Online]. Available: https://doi.org/10.1145/2661136.2661159

[31] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 311–320. [Online]. Available: https://doi.org/10.1145/1368088.1368131

[32] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines,"

in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 105–114.

[33] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 127–136, 2004.

[34] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *Software Engineering, IEEE Transactions on*, vol. 30, no. 6, pp. 355–371, 2004.

[35] B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos, "Towards a generic aspect-oriented modeling framework," in *Models and Aspects workshop, at ECOOP 2007*, July 2007. [Online]. Available: http://www.irisa.fr/triskell/publis/2007/morin07a.pdf

[36] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jézéquel, "Managing variability complexity in aspect-oriented modeling," *Model Driven Engineering Languages and Systems*, pp. 797–812, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87875-9_55

[37] S. Apel, C. Kästner, and C. Lengauer, "Language-independent and automated software composition: The featurehouse experience," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, pp. 63–79, 2013.

[38] S. Apel, C. Kästner, and C. Lengauer, "Language-independent and automated software composition: The featurehouse experience," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 63–79, 2011.

[39] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling automation and flexibility in product derivation," in *SPLC'08*. IEEE, 2008, pp. 339–348.

[40] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Software Product Lines: Going Beyond*, J. Bosch and J. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 77–91.

[41] F. Damiani, L. Padovani, I. Schaefer, and C. Seidl, "A core calculus for dynamic delta-oriented programming," *Acta Informatica*, vol. 55, no. 4, pp. 269–307, Jan. 2017. [Online]. Available: https://doi.org/10.1007/s00236-017-0293-6

[42] M. Erwig and E. Walkingshaw, "The choice calculus: A representation for software variation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 1, pp. 1–27, 2011.

[43] J. B. Ferreira Filho, S. Allier, O. Barais, M. Acher, and B. Baudry, "Assessing Product Line Derivation Operators Applied to Java Source Code: An Empirical Study," in *19th International Software Product Line Conference (SPLC'15)*, Nashville, TN, United States, Jul. 2015. [Online]. Available: https://hal.inria.fr/hal-01163423

[44] B. Combemale, O. Barais, O. Alam, and J. Kienzle, "Using cvl to operationalize product line development with reusable aspect models," in *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, 2012, pp. 9–14.

[45] T. Degueule, J. B. F. Filho, O. Barais, M. Acher, J. Le Noir, S. Madelénat, G. Gailliard, G. Burlot, and O. Constant, "Tooling support for variability and architectural patterns in systems engineering," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 361–364. [Online]. Available: https://doi.org/10.1145/2791060.2791097

[46] J.-M. Horcas, M. Pinto, and L. Fuentes, "Extending the common variability language (cvl) engine: A practical tool," in *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, ser. SPLC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 32–37. [Online]. Available: https://doi.org/10.1145/3109729.3109749

[47] G. G. Pascual, M. Pinto, and L. Fuentes, "Run-time support to manage architectural variability specified with CVL," in *Software Architecture - 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. Drira, Ed., vol. 7957. Springer, 2013, pp. 282–298. [Online]. Available: https://doi.org/10.1007/978-3-642-39031-9_24

[48] J. Font, L. Arcega, Ø. Haugen, and C. Cetina, "Achieving feature location in families of models through the use of search-based software engineering," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 363–377, 2018. [Online]. Available: https://doi.org/10.1109/TEVC.2017.2751100

[49] J. Krüger, G. Çalıklı, T. Berger, T. Leich, and G. Saake, "Effects of explicit feature traceability on program comprehension," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 338–349.

[50] W. Fenske, S. Schulze, and G. Saake, "How preprocessor annotations (do not) affect maintainability: A case study on change-proneness," in *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 77–90. [Online]. Available: https://doi.org/10.1145/3136040.3136059

[51] A. Rodrigues Santos, I. do Carmo Machado, E. Santana de Almeida, J. Siegmund, and S. Apel, "Comparing the influence of using feature-oriented programming and conditional compilation on comprehending feature-oriented software," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1226–1258, 2019.

[52] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the# ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[53] W. Mahmood, D. Strüber, A. Anjorin, and T. Berger, "Effects of variability in models: A family of experiments," *Empirical Softw. Engg.*, vol. 27, no. 3, may 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10112-3

[54] J. Echeverría, J. Font, O. Pastor, and C. Cetina, "Usability evaluation of variability modeling by means of common variability language," *Complex Syst. Informatics Model. Q.*, vol. 5, pp. 61–81, 2015. [Online]. Available: https://doi.org/10.7250/csimq.2015-5.05