Variant Management Solution for Large Scale Software Product Lines

Richard Pohl[1], Mischa Höchsmann[2], Philipp Wohlgemuth[1], Christian Tischer[1]

[1] Robert Bosch GmbH
Stuttgart, Germany
{richard.pohl|philipp.wohlgemuth|christian.tischer}@de.bosch.com

[2] itemis AG
Stuttgart, Germany
mhoechsmann@itemis.de

ABSTRACT

Application lifecycle management for large scale software product lines (SPL) comes with the challenge to integrate distributed development activities across different parts of an organization and the engineering process in a tool landscape. Variant management is a cross-cutting concern that has interaction points with many of those integrated solutions. At Bosch, two different tools are used for variant management: pure::variants, a feature modeling tool for describing the feature-oriented product decomposition, and the custom tool MIC that offers a more comprehensive set of fine-grained variability management mechanisms. These include parameters, automated configurations or constraints. In turn, it is more suitable for component selection that is done close to the technology. In this experience report, we present a methodological approach on how to use the two tools with a technical integration solution we developed. Its purpose is to serve as an example for establishing successful variant management in large-scale product lines with respect to methodology and tools.

Author Keywords

Software Product Lines; Variant Management; Application Lifecycle Management; Experience Report

ACM Classification Keywords

Software and its engineering~Software product lines • *Software and its engineering~Software development methods*.

1 INTRODUCTION

Starting in the early 2000s, Bosch introduced a software product line (SPL) [8] approach to meet increasing demands in highly cost intensive market segments in the automotive sector [10]. This approach was successfully applied to manage complexity due to an increasing number of variants of engine control software for ECU products [12]. These products are highly variant-intensive as they are used in various products, such as airbags, window lifters and driver assistance systems. Moreover, even the variation within one product is high in the automotive industry due to mass customization, leading to up to 15,000 variation points. In addition to products, the development processes vary inside an organization like Bosch that has smaller business units with only 100 active developers as well as business units with up to 9,000 active developers.

Nowadays, application lifecycle management (ALM) [4] tools have become state of the art and feature-based variability modeling tools are available for variant management. Such tool support is crucial for maintaining productivity in today's complex development processes and environments This includes activities such as requirements development and management, design management, change management, source code management and quality management (see Fig. 1). Known advantages of ALM tools are: automation throughout the process for keeping traceability links, reporting, and collaboration. In this context, variant management is a cross-cutting activity that deals with all process steps. Thus, variant management approaches and tools need a very high degree of integration with existing ALM tools.

Variant management is split into two different aspects covering different views on the system: problem space and solution space. Problem space concepts are managed with a feature-model based approach that relates features to problem-space artifacts such as customer requirements. A state of the art tool that we use for this is pure::variants [11]. However, solution space variant management is more complex, especially when dealing with highly distributed embedded systems as in our ECU product lines. Software and technology add variability and constraints that have to be fulfilled. Thus, variant management in solution space cannot be reduced to manual feature selection. In particular, automated configuration is not just followed by simply deriving configurations throughout all levels of granularity in the system. Instead, configurations for the whole system throughout all levels have to be derived based on different factors. Such factors include parameters of configurable third-party modules, parameters of the used hardware and the selection of data and instructions for the used hardware. In our case, solution space configurations are derived based on

ICSE-SEIP '18, May 27-June 3 2018, Gothenburg, Sweden
© 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183523

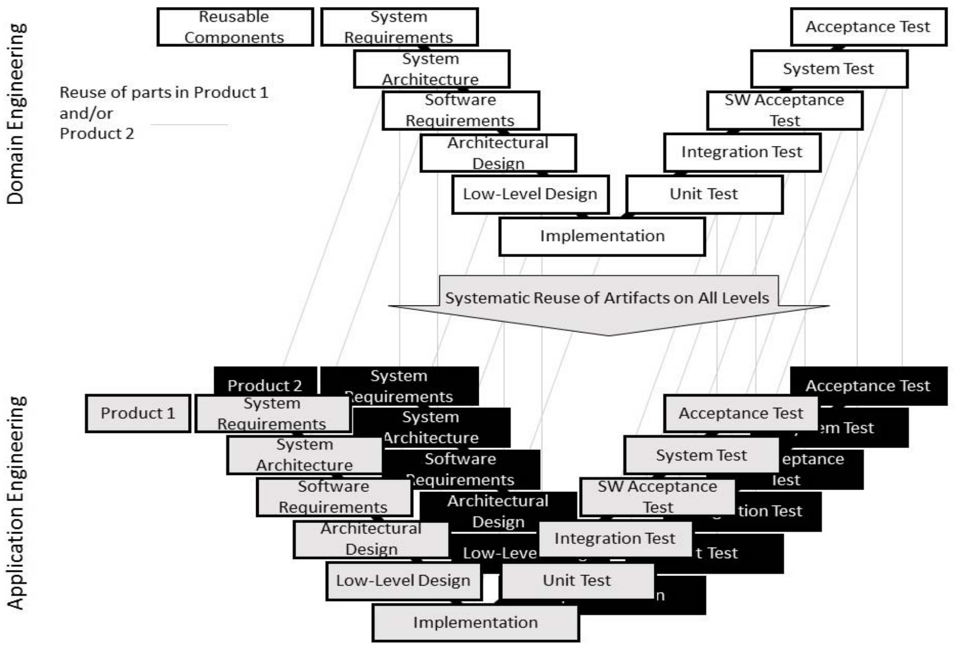


Figure 1. Reuse across the system development lifecycle

configuration constraints, partial configurations and parameters. This is handled by our tool solution MIC.

This experience report presents our MIC tool and shows how the tools pure::variants and MIC fit into the established ALM tool landscape at Bosch. Section 2 summarizes the theory of SPL engineering and the challenges of developing large-scale PLs in large organizations. Section 3 explains the concepts of problem space and solution space and highlights different challenges of development Section 4 introduces feature models and presents pure::variants as a feature modeling tool. Section 5 presents our solution-space variability tool MIC. Section 6 shows how pure::variants and MIC are integrated. Section 7 presents first experiences with variant management at our ECU projects. Section 8 summarizes some of the related work. In our final section 9, we summarize our report and present our conclusions.

2 THEORY OF SPL ENGINEERING AND THE CHALLENGES OF DEVELOPING LARGE-SCALE PRODUCT LINES IN LARGE ORGANIZATIONS

Large organizations developing products in the automotive industry are facing more and more the challenge to react to the customer's needs in a fast and cost-efficient way, while the product diversity and complexity is rapidly growing. A well-known approach to achieve shorter development cycles and support high product diversification is to identify and develop common solutions for reuse in a large variety of product variants. The goal is to avoid multiple development efforts for new product variants by reusing already developed components "as is" or with only minor changes. This methodology is commonly referred to as product line engineering (PLE) or specifically in the context of software development as *software product line* (SPL). The main focus of this paper is on software, even if automotive products always consist of hard- and software.

Developing and operating a SPL requires an organization wide agreed business strategy and development approach. The business strategy needs to consider which markets and products shall be addressed, thus basically defining the features scope and requirements to be considered. The SPL development approach consists of two basic activities, as shown in Fig 1:

1) Develop components for reuse:

Stakeholder requirements are defined in cooperation with customers or marketing divisions. Commonalities and variability in the requirements need to be evaluated and allocated to the elements of a suitable system and software architecture, e.g. sub-systems and software components. Design and development of the respective software components need to implement solutions that enable reuse of all parts addressing common requirements, while providing configuration capabilities for the variant parts. All requirements need to be linked to the implemented components and the included configuration points. All components, requirements and other development artifacts available for reuse need to be managed in a common data base.

2) Develop products with reuse:

Once the decision is taken to develop a new product or product variant reusing existing components from the asset base, the right components and configuration settings have to be identified. Mapping the requirements for a specific product to the available features in an asset base and identifying missing solutions is one of the most challenging task in a SPL. Based on this analysis, the actual product construction starts: implementation of missing requirements, integration and configuration of the reused parts. To support the final system testing and verification, reusable assets need to provide test cases related to the configured features.

In large organizations, these activities are repetitive: feature scope and asset base need to be continuously maintained, product derivation runs in parallel for multiple product variants at a time, using different versions and releases of the reusable artifacts, see Fig. 2

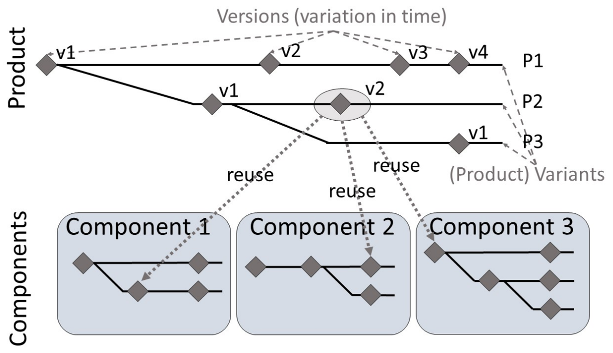


Figure 2. Product variation in time and product design space

From a top-down perspective - considering business level down to implementation level - managing the product features and their relations are a crucial aspect of SPLs. Product feature dependencies and their relations to development artifacts need to be considered in a vertical way (e.g. a system requirement satisfies a customer requirement) as well as in a horizontal way (e.g. a requirement is tested by a test case). These vertical and horizontal links can either be valid in the context of one single product variant only, in several variants or in all variants. Additionally, there is a need for managing all these *configurations* (i.e. configured and reusable product-variants) and provide some additional information (metadata) about all these configurations in order to easily identify them for their intended use. Moreover, other constraints such as specific dependencies to other configurations of other components or sub-systems of the product can be of interest.

An established approach for describing product features and their dependencies are feature models. They support understanding the complexity of a product as well as sharing this knowledge among different stakeholders like customers, project managers and developers.

3 THE VARIANT INTERFACE BETWEEN PRODUCT MANAGEMENT AND SYSTEM DEVELOPMENT

Product managers are focusing on customer relevant *characteristics of a product*, commonly referred to as features. Feature models represent those relevant features along with rules specifying valid feature combinations. This perspective on variants is called *problem space*. It explicitly excludes considerations on how products are being realized. The latter will be handled by all the system development domains (SDD, referred simply as "domain" in the following) involved (e.g. requirements engineering, design, implementation, etc., see Fig. 1). Variation points on system development level represent *technical variation points* within the specific development domain. This perspective is called *solution space*.

3.1 Integration of problem and solution space

Managing both perspectives to variant handling leads to the need to integrate both views when deriving specific variants. The high level objective of the integration is to tailor specific product variants in an automated way based on a set of feature selections. All subordinate domain artifacts should be assembled and configured automatically based on rules referring to features, the feature mapping.

The major challenge in tool development is to provide the glue between the feature model (problem space) and the technical variation points (solution space). To separate the tool terminology from the conceptual terms problem- and solution space, we are referring to variant management tools (VMT) representing the problem space and system development tools (SDT, referred simply as "tools" in the following) for the solution space respectively, as defined by the variability exchange language (VEL) [9]. While there will be just a single VMT, each domain in Fig. 1 owns its individual, dedicated, tooling. So individual integration approaches are needed. The aspects determining the integration effort or whether an integration is applicable at all are varying for each specific domain. Nevertheless, the following integration steps are common to all domains:

1. Providing feature mappings
2. Transforming the domain-specific variation points into a technical representation, supported by the variant management tool

A feature mapping describes the activation/configuration of variation points in the tools based on feature selections made in the VMT. Features are considered being more stable than technical variation points. Accordingly, the direction of the mapping usually tends to be from tools to VMT, i.e. the tool must be enhanced by feature mappings. Applying the feature mappings is the responsibility of the VMT. This is the reason for the second integration step: Transforming the tools' variation points in an abstract representation. It is supported by the VMT.

3.2 Variability Exchange Language

As part of the research project SPES_XT (Software Platform Embedded System 2020, [8]), supported by the German Ministry of Education and Research (BMBF), the variability exchange language has been specified. In a nutshell, the VEL is an abstract and formal representation of variation points aiming to provide a unified standard for the exchange of variant information. It is a perfect match for the integration scenario mentioned previously as it provides support for both directions: from the tool to VMT and, after feature selection, back from the VMT to tool, see Fig. 3.

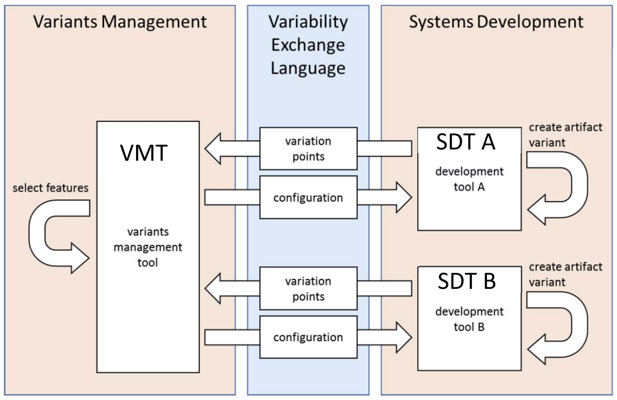


Figure 3. Variant information exchange using VEL [9]

3.3 Capabilities

The tool architecture given does not only support the initial goal of automated creation of product variants, it can also be used to identify unmapped variation points on both sides. In more detail, it allows

(1) Identifying combinations of customer features which are not yet supported by the underlying platform and vice-versa.
(2) Identifying technical variations, which aren't used in production yet.
(3) Annotating the solution space variation points with additional non-functional attributes (e.g. costs) being considered as a further discriminator when there are several solutions available to realize a certain product variant.

The following chapters show the tool integration of both, the problem and solution space, based on the exemplary solution domain software development.

4 PROBLEM SPACE: VARIANT MANAGEMENT TOOL PURE::VARIANTS

As already mentioned, one aspect of variant management is to have the possibility to describe a product which is going to be developed in an abstract way without focusing on how the product is going to be realized. In the *problem space*, this is done by describing the product with features (e.g. describing the product "car" available with different engine types or different options of driver assistance functionality). In addition, rules like relations between features, constraints and restrictions can be specified which are then later used to determine which feature combinations are valid to form a valid product variant. In order to be able to derive all potential product variants, a superset of features have to be specified (i.e. for the whole product family) this is the so called *150% model* of a product line.

The description of the features and their relations to each other are done in a feature model. Later, the features in the model are mapped to artifacts in the system- and software-development-domain (e.g. requirements, design elements, source-code files) the *solution space*. At Bosch, *pure::variants* developed by Pure-Systems [11] has been chosen as the variant management tool covering exactly the above described capabilities (feature-modelling and support of mapping the features to all the software development lifecycle artifacts).

Pure::variants is one of the few feature modelling tools on the market which offers full-fledged support of connecting to different systems or software domain tools used for the different disciplines along the V-cycle such as requirements-management, architecture or test-management tooling. Moreover, it also features high extensibility to other system development domain tools via an API for Java or using the VEL as a way to describe variation points which in turn can be then imported into pure::variants to provide the mapping between the features in the feature model and the artifacts in the solution space.

4.1 Feature Model – the Description of the Problem Space

In pure::variants *feature models* are comprised of features in a hierarchy which form a tree like structure. In this structure, features are either represented as mandatory, optional or an alternative out of a list of features. Furthermore, the features can be enriched with rules describing relations between one feature and another feature (e.g. "feature A excludes feature B" or "feature A implies Feature C") or other restrictions and constraints. Those restrictions and constraints are typically used to test some condition under which a feature becomes a valid selection or to test whether an attribute (a key value pair which can be attached to a feature) has the right value. Figure 4 shows a simple feature model schema.

4.2 Feature Mapping – the Family Model

As a next step, the features in the problem space have to be mapped to the artifacts in the solution space, thus creating a *family model*. In this section a general approach is described. In the next chapters the mapping between features and the solution space artifacts will be described in detail.

The mapping is normally done in the respective system or software development tool. For instance, a couple of requirements are mapped to a specific feature in the feature model. Pure::variants offers a small extension which lets one access the corresponding feature model and its feature names that can be used in an attribute or as a system constant in the system or software development tool or source code.

Fig. 4 shows an example for a simple feature model depicting some of the features found in a car. The car has the two mandatory features "Engine Type" and "Gear Box". For either one, a particular child feature can be picked from a set (alternative group). Moreover, there are optional features "Start-Stop Automatic" and "User Driving Mode Selector" that can be part of a product or not. For the latter, any combination of features from or-group of different driving modes can be picked. The "requires" constraint between "Start-Stop Automatic" and "Gasoline Engine" means that "Start-Stop Automatic" is only provided in gasoline cars.

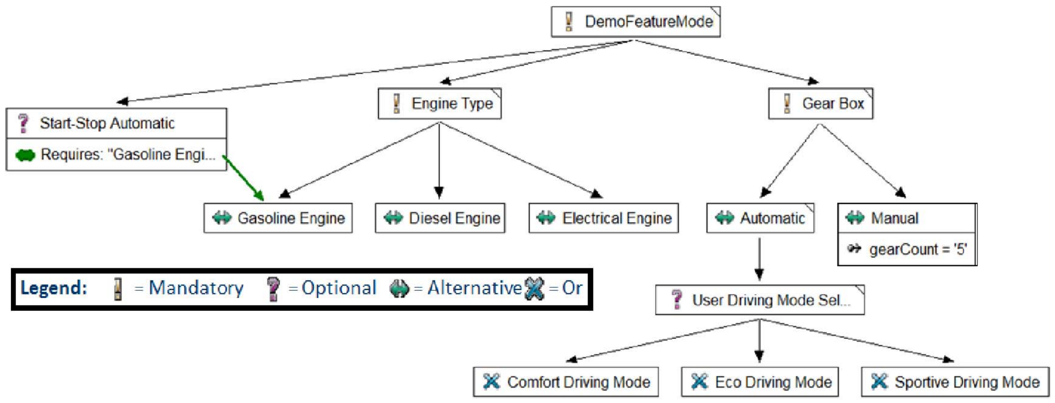


Figure 4. Simple feature model schema

4.3 Configuration of a Variant – The Variant Model

When the feature model and the mapping to the solution space artifacts are defined, a *100% model* of a product variant can be configured. This is done through a *variant model* which allows to explicitly select features a specific product variant. When selecting a feature, pure::variants will check relevant relations, constraints and restrictions. The tool will either raise an error-message, select missing dependent features or will exclude other features from the selection if a constraint or restriction is violated. As soon as a variant has been configured, a *transformation* can be used to derive the variant in the solution space.

4.4 Benefits

By using such a feature modelling tool for variant management, every project role who needs deep insight into the system being developed, can benefit from its capability to make the system less complex. It provides an abstraction on what features are contained in the product and the dependencies between them (i.e. an abstract and simplified view on the product's architecture). Also, the 150% to a 100% derivation of variants with variant management with the help of feature modelling tools saves a lot of time and money when developing new product variants and even new products.

5 SOLUTION SPACE: SOFTWARE INTEGRATION AND ARTIFACT BASED VARIANT MANAGEMENT

Like systems development, the process of software development can be broken down into different development domains. Accordingly, there will be several areas of variant management for software development. Without any claim for completeness, Figure 5 highlights these areas.

Subsequently we will focus on a specific solution for the area of `Artifact Selection` as one part of the common Bosch variability concept. The conventional way to select variant specific artifacts is to manage simple file lists manually. File lists neither provide a sophisticated variant mechanism nor a concept for partitioning in means of hierarchical software reuse. Facing a number of hundreds, even up to thousands of components the manual management of file lists is hardly an option.

5.1 MIC - Artifact Based Variant Management

In the following, we will focus on the *artifact based* variant management approach applied at the Robert Bosch GmbH. The framework in use, realizing artifact based variability in the solution space, is an in-house solution called metadata information container (MIC). The main purpose of MIC is to support a wide variety of software and documentation build processes by providing metadata and variant information for all build relevant artifacts in the software project. The variant calculation can be seen as filter removing all artifacts which are not part of a specific variant. The term artifact refers to resources, more precisely files and folders. Variation points inside files, like the conditional execution of code at run- or compile time, is not in scope of MIC.

Enabling a maintainable vertical integration of software components (i.e. SW-reuse) is the guiding design principle for the MIC framework. The MIC variant mechanism is coping with the software reuse use case by forcing strictly encapsulated, ready-to-use components in means of variants. Such components, offering a client variant interface, are named variant configuration unit (`VCU`).

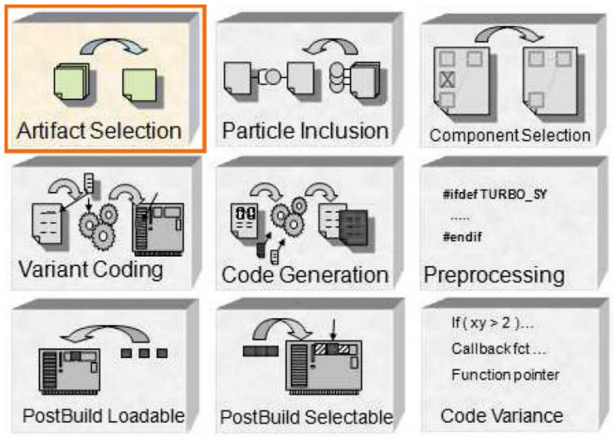
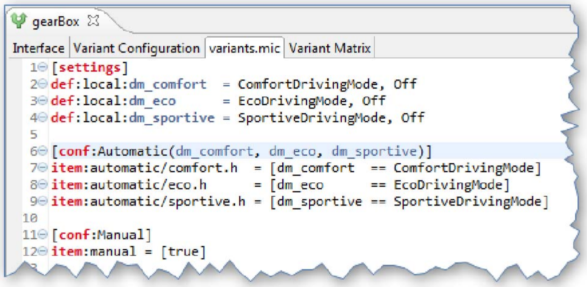


Figure 5. Classification of variant management mechanisms in the software development solution space

Typically, the major part of a project is configured hierarchically, whereas parent VCUs select the variation points (i.e. resolving variants) of the child VCUs included. Thus MIC supports both, the definition and the selection of variants. MIC offers a simple domain specific language, which can be split into two major aspects. One to define metadata information for all project resources and one to specify variants. Depending on the project needs, both aspects can be used exclusively or in combination. The specification of metadata and variants is persisted in files with a *.mic file extension, right next to the software part it is related to.

5.2 Variant Configuration Unit (VCU)

The following snippet shows a brief example of a variant specification for a VCU (i.e. a component exposing variation points to clients).



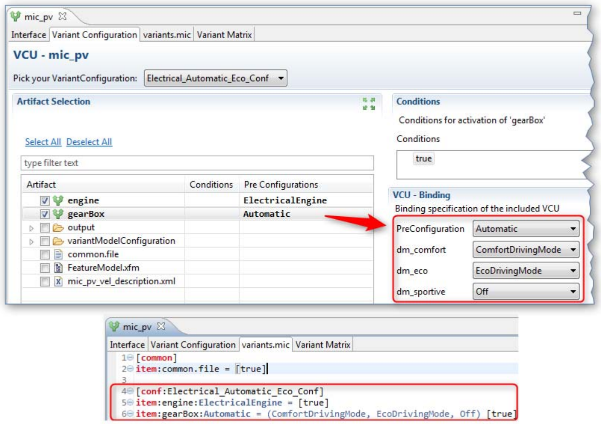
```
1  [settings]
2  def:local:dm_comfort  = ComfortDrivingMode, Off
3  def:local:dm_eco      = EcoDrivingMode, Off
4  def:local:dm_sportive = SportiveDrivingMode, Off
5
6  [conf:Automatic(dm_comfort, dm_eco, dm_sportive)]
7  item:automatic/comfort.h  = [dm_comfort  == ComfortDrivingMode]
8  item:automatic/eco.h      = [dm_eco      == EcoDrivingMode]
9  item:automatic/sportive.h = [dm_sportive == SportiveDrivingMode]
10
11 [conf:Manual]
12 item:manual = [true]
```

Figure 6. VCU - Variant specification for component gearBox

The VCU *gearBox* shown in Figure 6 gives an overview of the basic internal and external variation points available at MIC. The component itself can be seen as a first optional variation point as it can be either used by clients or not. In line 6 and 11 follows the definition of two predefined, named, variants. Clients including this VCU need to choose between one of these two PreConfigurations and provide arguments for the parameters defined (e.g. *dm_comfort).* With the selection of a specific PreConfiguration all resources mentioned within are considered active for the product variant currently configured. Child-VCUs are resources too, so the process of configuration is hierarchical. Finally, line 7/8/9 shows implicit internal variation points, conditionally activating source files.

PreConfigurations are the major design element to form a component's client interface, each representing one possible variation of that component. Also PreConfigurations are aggregating the definitions of all required internal resources and the selection of further variation points which might exist in child components.

Integrating the gearBox VCU into any surrounding context (i.e. project or component) means to bind the variations defined, in our case one of the PreConfigurations and the related arguments, if defined. Figure 7 shows such an integration scenario. Both views are showing the content of the project level MIC configuration, at the top in a graphical editor and below based on the raw textual representation the graphical editor is based on**.**



VCU - mic_pv

Pick your VariantConfiguration: Electrical_Automatic_Eco_Conf

Artifact	Conditions	Pre Configurations
✓ engine		ElectricalEngine
✓ gearBox		Automatic
output		
variantModelConfiguration		
common.file		
FeatureModel.xfm		
mic_pv_vel_description.xml		

Conditions for activation of 'gearBox'

Conditions: true

Binding specification of the included VCU

PreConfiguration	Automatic
dm_comfort	ComfortDrivingMode
dm_eco	EcoDrivingMode
dm_sportive	Off

```
[common]
item:common.file = [true]

[conf:Electrical_Automatic_Eco_Conf]
item:engine:ElectricalEngine = [true]
item:gearBox:Automatic = (ComfortDrivingMode, EcoDrivingMode, Off) [true]
```

Figure 7. Project level configuration for variant "Electrical_Automatic_Eco_Comf"

The green icons shown in the tree of resources representing VCUs, exposing an own variant interface which need to be configured properly when included. All other resources are handled as regular artifacts (files and folders) and can be included in the current configuration without any restrictions.

5.3 Global Aspects

The example shown in Figure 6 is using a bottom-up approach since the origin of the variation points given (the defined parameters, e.g. dm_comfort) lays within the VCU. In real work projects there are also aspects which have an impact on many or all components at a time. For example, the version of the reference architecture or the hardware platform used. This is what we refer to as top-down approach and MIC supports it by applying so called global variant selectors. Global variant selectors are used in conditional selection statements inside VCUs. This includes a hard dependency from a VCU to a global set of variant selectors. While being handy in means of a simple configuration, the upstream dependency limits the reusability of components and the global set of variant selectors need to be managed by a superior instance.

5.4 Interaction between MIC and its Clients

MIC provides interfaces to several development tools, in the following called clients. Clients are not limited to read metadata information via the MIC interface but can also dynamically add new resources (and metadata) to existing variants which makes MIC a perfect fit for the continuous integration process. The input for software or documentation build processes are determined by querying the MIC system on relevant metadata information for specific variants. At the same time, artifacts which are created by the build process need to be registered at MIC in order to make them available to subsequent build steps. MIC supports that incremental classification process by the temporarily registration of generated artifacts.

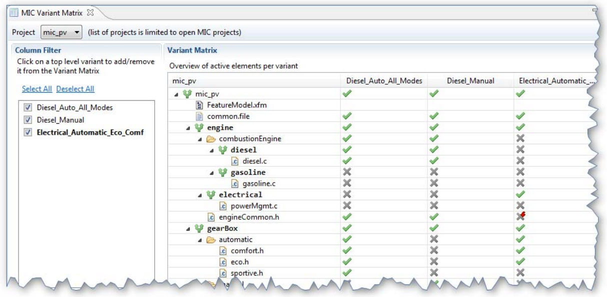
5.5 Variant Design

Using MIC, component architects can decide which variation points are going to be part of the component interface and which ones should be hidden internally to hide/reduce complexity. In addition, they can codify configuration know-how by aggregating valid combinations of child components in a `PreConfiguration`. Ideally, the integrator finally selects the first level components only and makes a choice on the (high level) variation points present by using a guided selection dialog (see Figure 7).

5.6 Supporting the Application Lifecycle

One major challenge for the automotive industry addressed by ALM is *traceability*. MIC covers the legal claim for traceability as given in the ISO 26262 [6] by back links from resources to the related variant specification. MIC fulfills the needs to serve as a general tool applied across the whole vertical chain of integration. It offers graphical support for both, editing the variant specification (Figure 7) and verifying the variant calculation result (Figure 8).

With respect to the software reuse use case, MIC offers a mechanism to temporarily overrule the set of active resources in a non-invasive way (i.e. without changing the VCUs itself). This kind of patch mechanism is required to enable a fluent workflow, especially when external components used (i.e. component managed by other departments) are erroneous or just do not fit the project needs yet. The tooling will indicate such patched elements by a small red arrow decorator like shown in the third column "*Electrical_Automatic…*" of Figure 8 which indicates that the file *engineCommon.h* has been removed from that particular variant by the overruling mechanism mentioned.


MIC Variant Matrix

Project mic_pv (list of projects is limited to open MIC projects)

Column Filter
Click on a top level variant to add/remove it from the Variant Matrix
Select All Deselect All
☑ Diesel_Auto_All_Modes
☑ Diesel_Manual
☑ **Electrical_Automatic_Eco_Comf**

Variant Matrix
Overview of active elements per variant

mic_pv	Diesel_Auto_All_Modes	Diesel_Manual	Electrical_Automatic_...
mic_pv	✓	✓	✓
FeatureModel.xfm			
common.file	✓	✓	✓
engine	✓	✓	✓
combustionEngine	✓	✓	✗
diesel	✓	✓	✗
diesel.c	✓	✓	✗
gasoline	✗	✗	✗
gasoline.c	✗	✗	✗
electrical	✗	✗	✓
powerMgmt.c	✗	✗	✗
engineCommon.h	✓	✓	✗
gearBox	✓	✓	✓
automatic	✓	✗	✓
comfort.h	✓	✗	✓
eco.h	✓	✗	✓
sportive.h	✓	✗	✗

Figure 8. Variant Matrix

6 INTEGRATION OF PURE::VARIANTS AND MIC

So far the tools pure::variants and MIC have been considered separately, with pure::variants as a general variant management tool and MIC as a specific variant solution for resource management. Now we are going to focus on how both tools can be integrated with our ALM system. In the following, numbers in brackets (x) refer to Fig. 9.

6.1 Objectives

The basic assumption on all (or at least most of the) variation points in the solution space is that they are somehow related to customer relevant features modeled in the feature model (1) in the problem space. They will finally realize the required features of a specific product variant. With respect to the tooling, the relation of product features and platform variants lead to the need to couple both of them, building a seamless integrated toolchain. This way the variant configuration for the solution space can be derived automatically from concrete variant descriptions (3) in the problem space. The integration also aims to grant architects and integrators an overview of the 'problem demand' and the corresponding 'solution supply' in a formal way.

6.2 Integration Scenario

The basic idea of the integration is to generate new product variants at the root level of MIC automatically, based on a given variant description model (3). The main integration junction points are called "Propagate" making the MIC variation points (product line) available in pure::variants and "Generate" to automatically create a new product variant in MIC. For both directions we are utilizing the VEL as formal, abstract representation of the variant information exchanged.

Figure 9 gives orientation on all the model artifacts involved. Elements in the upper half of the image are related to the whole product line (i.e. no specific variant has been derived) while the lower ones are referring to specific product variants. Similarly, the elements at the right hand side of the vertical line are related to a specific domain while the left hand ones are shared across several or all domains and thus will be present only once, valid for all domains.

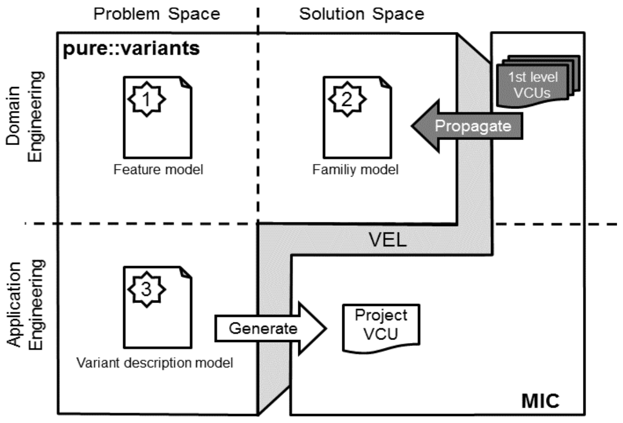


Figure 9. Coupling of pure::variants and MIC

6.3 Feature Mappings

Unlike variation points, feature mappings are not a basic MIC concept. Feature mappings serve the sole purpose of binding variation points to features (and thus creating a link/dependency from the solution to the problem space). We decided to start with just two different kinds of feature mappings.

First, we introduced a simple name based mapping for all `PreConfigurations` and `Parameters`. Applied to the example shown in Figure 8, the `PreConfiguration` *Automatic* would be activated when there is an active feature with the same name, and the argument *dm_comfort* would also be derived from the existence of a *ComfortDrivingMode* feature.

Second, for the activation of a whole component (VCU) we introduced a more flexible feature mapping which is about attaching a VEL-expression to the components metadata. The expression will neither be interpreted by MIC nor by the VEL transformation but passed through pure::variants. This way all expressions supported by pure::variants can be applied, for example native VEL-expressions or AUTOSAR-expressions. To give an example, an expression to activate a component based on the selection of one of the following two features *ESP*, *ABS* would look like:

```
    vel:or-feature-condition = ESP, ABS
```

6.4 Propagate MIC Variation Points

The first step of the integration is to fetch all the MIC variation points and feature mappings available and transform them into a VEL model. VEL already supports both concepts (variation points and feature mappings) so with the known semantic of the MIC variation points given, this conversion is straight forward. At that point the VEL model is called a `variationpoint-description` since it describes a whole product-line.

When collecting the variation points to be propagated, we collect all 1st level VCUs only. The term 1st level refers to the first VCUs found when traversing the component hierarchy top-down. Once the VEL model has been created it will be imported into pure::variants, represented by a family model (2). Pure::variants supports the VEL natively so this step works fully automated. From then on pure::variants can be used as usual, e.g. validating variant description models (3) against the family model (2).

Figure 10 shows all the technical steps required to derive a specific variant project. Assuming the feature model and the platform project are already present, the box "`specific product variant`" represents the only action to be conducted manually in order to derive a product variant. All other steps are fully automated.

6.5 Generating MIC Configuration

The second integration step will add a new product variant at the most upper MIC variant configuration. It contains selection statements and variant bindings according to the information given by the VEL model for all the 1st level VCUs collected in the first step. The VEL model itself can be automatically exported by pure::variants. Compared to the VEL model which has been used in the first step, the VEL model generated is based on the variant description model (3) and now contains concrete selections for all the variation points in addition. It represents a specific product variant.

6.6 Summary of the MIC Approach

The integration approach shown can be seen as a reference implementation. The integration of other domains will follow the same principles, including the utilization of the VEL. Nevertheless, the mapping of variation points and the integration of feature-mappings is a highly specific, conceptual task to be solved individually for each domain.

An aspect not covered by the integration scenario is the overlapping of features and variation points. From a technical viewpoint, a feature is a possible variation of a variation point. So the decision whether to place variation points or constraints in pure::variants or MIC is an explicit design decision with an impact on the benefits received.

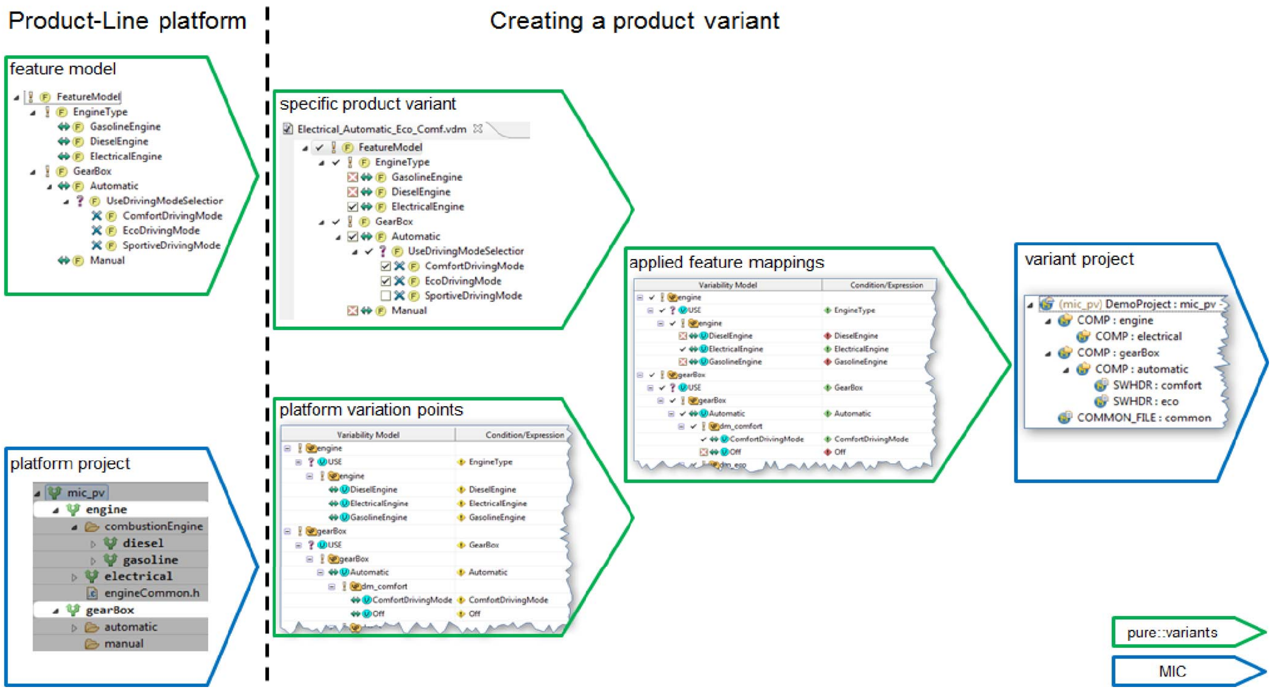


Figure 10. Steps to derive a specific product variant

7 Experiences of Variant Management in Customer Projects

Managing variants properly plays a crucial role in the domain of ECU development. Shrinking product development cycles, the high effort for verification and the benefits achievable when reusing a vast amount of assets developed, are the major drivers for variant management. At the same time, variant management has a relatively low degree of maturity when compared to other technical management domains, for example the defect management. Thus we are still at an early stage of applying such a holistic variant management approach. Even though it appears very promising to us there are no reliable empirical data on the benefits or opportunity costs yet. In the following we will give a brief overview of variant management aspects we have identified being the most challenging ones, technically and from organizational perspective.

7.1 Organizational Challenges

Variant management is a **cross-cutting** concern affecting both, the vertical and horizontal integration in a pervasive way. Cross-cutting does also mean that for most development steps there is not a single, dedicated tool to cope with variants. Instead, variant support is more like an add-on for established tools. Only when defining the impact of a variation point on the artifacts throughout the development cycle (e.g. requirements, tests, source code, calibration parameters, service needs …), it will support product development properly. So even variant support needs to be implemented individually per tool, all tools need to speak a common language (the feature model) to provide a holistic approach. Due to the high number of tools and departments involved, such a global approach must be rolled out by a central department, backed up by high management commitment.

Introducing a unique feature model for a product-line is a challenge of its own. Historically most tool domains manage an individual subset of features, like the different departments do alike. When consolidating those different feature models into a single one, the different abstraction levels of features must be kept but still be interconnected. We accomplished that by introducing hierarchical feature models. The process of resolving conflicts when unifying features, previously defined ambiguously, is what we call **feature clearing**. As to our experience, such a central governance process needs strong roles in the organization: it can either be done by product management defining the major capabilities of the product, or by system architects, who additionally can consider major realization implications of a new feature. In large organizations setting up such roles and processes needs strong management commitment to align all affected parties

7.2 Variant Reuse and Encapsulation

After rolling out the approach across different departments the aspect of software reuse came into focus. The strong encapsulation of MIC prohibits the modification of software deliveries in means of variants. For strictly separated producer / consumer relationships this is exactly the intended behavior. But it also turned out that the strict encapsulation is blocking more collaborative departments from adjusting software components to their needs.

The two contrary forces identified here are the intention of the producer to limit the use of a software (component) to a predefined degree, versus the potential need of the consumer to adjust the software, exceeding the degree foreseen by the provider. Indeed, all assets which will be reused in some way, need to resolve that conflict somehow. In our case we decided to resolve it by loosening the tool constraints and introducing processual constraint on the other side, if required.

7.3 Research Demand

Research activities in the field of product lines and variant management seems to lack a **proper categorization** of variants. There might exist some criteria of variants which could help identifying proper management solutions, or whether a potential variant is considered being a variant at all depending on the context of work. Whether a variant has been foreseen (i.e. an intended feature of a system/platform) or it occurred unpredictably (unmanaged/ad-hoc variant) could be such criteria.

While interdependencies between features on the same abstraction level could be handled properly, the net of formal dependencies is growing fast and getting more complex when connecting features of different abstraction levels. To give an example, it is quite straight forward to define that cars manufactured for the European market must have a steering wheel on the left hand side of the car. However, defining the dependency between the region and the required amount of physical memory is way more complex. There are a lot of technical, numerical variants where it might just not be feasible to provide a full mapping to high level features. Again, a categorization of variants (e.g. by abstraction level) could help finding proper abstraction levels used to divide variant management domains in order to reduce complexity and build individually tailored solutions.

8 RELATED WORK

SPL engineering approaches were applied by various organizations from different work areas [12]. Wölfl et al. [2] present their experiences on introducing an SPL approach in the avionics sector at Airbus Helicopters. Their approach has some similarities in the setup such as the need for a proof of functional safety induced by according standards. The main focus of their work is on combining verification and validation assets with the source code in order to reduce the effort in obtaining the qualification required for this functional safety proof.

Cortiñas et al. [1] present the introduction of a SPL approach for geographic information systems (GIS) in a small-to medium enterprise. They cover in particular the domains requirements analysis and architecture design and establish a way of deriving the application artifacts. Their approach is currently relying on a high manual effort.

Benavides et al [5] present the variability management approach used by the tax agency of the central government of Ecuador. Their approach is asset-based and relying on a common architecture for all products. The focus of their work lays on providing first-class variability modeling mechanisms and tools for making the software configurable throughout all parts. Noticeable is that they also see the need to separate the logical feature model from the actual configuration of assets. Even though they do not depend on hardware-implied constraints, the IT infrastructure leads to technical constraints introducing solution space variability.

Manz et al. [3] present experiences from an integrated variant management approach applied within the automotive industry at Daimler. Their approach is based on an integrated feature model that is split into several feature models for each development phase. The main challenge of their approach is that they require specialized tooling for integrating the feature models and that standardized interfaces for variability exchange are needed. In our approach, these challenges have been partly addressed by our MIC tool and the Variant Exchange Language VEL.

9 CONCLUSION

In this experience report, we present the current approach for SPL-based variant management approaches at Bosch BBM. For supporting variant management, two different tool solutions were used. One is pure::variants, a feature modeling tool that assists the users from problem space such as product management and early requirements analysis. The second tool is a custom tool MIC that is used for managing the solution space for efficient component selection.

Variant management is a cross-cutting concern in the engineering process. Comparing the current setting with the setting one and a half decades ago, in today's engineering process our variant management tools have to be integrated in an ALM tool landscape covering the whole process. This yields questions of interactions between the tools that go beyond simple binding of variants. Along with these tool enhancements, central governance processes and strong roles have to be established to manage the product features and cross cutting realization implications

This report presents a methodological and a tooling approach working with the two tools in an integrated manner. With this we are capable of efficiently managing variants at different levels of granularity. The presented ideas may serve as a basis for evaluating solutions for variant management and provide valuable advice for understanding how leading-edge variant management can be efficiently introduced in a modern distributed software engineering process supported by ALM tools.

REFERENCES

1. Alejandro Cortiñas, Miguel R. Luaces, Oscar Pedreira, Ángeles S. Places, and Jennifer Pérez. 2017. Web-based Geographic Information Systems SPLE: Domain Analysis and Experience Report. In *Proc. 21st International Systems and Software Product Line Conf.* (SPLC '17), ACM, New York, NY, USA, 190-194.
2. Andreas Wölfl, Norberd Siegmund, Sven Apel, Harald Kosch, Johann Krautlager, Guillermo Weber-Urbina. 2015. Generating Qualifiable Avionics Software: An Experience Report. In *Proc. 30th Int'l Conf. on Automated Software Engineering (ASE'15).* IEEE, Lincoln, NE, USA.
3. Christian Manz, Michael Stupperich, and Manfred Reichert. 2013. Towards Integrated Variant Management in Global Software Engineering: An Experience Report. In *Proc. 8th Int'l Conf. on Global Software Engineering*. IEEE, Lincoln, NE, USA.
4. D. Chappel. 2008. What is Application Lifecyle Management? Technical report, David Chappel & Associates, Retreived from http://www.davidchappell.com/whatisalm-chappell.pdf.
5. David Benavides and José A. Galindo. 2014. Variability management in an unaware software product line company: an experience report. In *Proc. 8th Int'l Workshop on Variability Modelling of Software-Intensive Systems* (VaMoS '14). ACM, New York, NY, USA.
6. International Standardization Organization. 2011. *Road vehicles – Functional Safety – Part 10: Guideline on ISO 26262.* ISO, Geneva.
7. Klaus Pohl, Günther Böckle and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin/Heidelberg.
8. Klaus Pohl, Manfred Broy, Heinrich Daembkes, Harald Hönninger. 2016. *Advanced Model-Based Engineering of Embedded Systems.* Springer, Berlin/Heidelberg
9. Martin Große-Rhode, Michael Himsolt and Michael Schulze. 2015. *The Variability Exchange Language (Version 1.0).* Retrieved from https://www.variability-exchange-language.org/
10. Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. 2004. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proc. Third Int'l Conf. on Software Product Lines (SPLC'04).* Springer, Berlin/Heidelberg.
11. Pure-Systems GmbH. 2006. Variant Management with pure::variants, Technical White Paper. Retrieved from https://www.pure-systems.com/mediapool/pv-whitepaper-en-04.pdf
12. SPLC.net. 2009. *Software Product Line Hall of Fame*. Retrieved from http://splc.net/fame/bosch.html