Contract number: ITEA2 – 10039

Safe Automotive soFtware architEcture (SAFE)

ITEA Roadmap application domains:

Major: Services, Systems & Software Creation

Minor: Society

ITEA Roadmap technology categories:

Major: Systems Engineering & Software Engineering

Minor 1: Engineering Process Support

WP4

Deliverable D3.4.b

Guideline for description of variant management techniques

Due date of deliverable: 30/07/2014

Actual submission date: 30/07/2014

Start date of the project: 01/07/2011 **Duration:** 36 months

Project coordinator name: Stefan Voget

Organization name of lead contractor for this deliverable: pure-systems GmbH

Editor: Michael Schulze

Contributors: Michael Schulze (pure-systems GmbH), Markus Oertel (Offis), Thomas Peikenkamp (Offis), Nico Adler (FZI), Martin Hillenbrand (FZI)

Revision chart and history log

Version	Date	Reason
0.1	01.09.2012	Initial Draft Version 3.4a
0.2	06.02.2013	Variant management using PREEvision
0.3	07.02.2013	Variant management using pure::variants
0.4	12.02.2013	Two section added to approaches for providing variant management
0.5	19.02.2013	Draft version of motivating example
0.6	22.02.2013	Configuration Package description
0.7	25.02.2013	Application of proposed approach onto example
0.9	25.02.2013	Review version
1.0	27.02.2013	Review comments incorporated; Final version
1.1	30.07.2014	Final Version 3.4b

1 Table of contents

2 List of figures

3 Executive Summary

The deliverable is targeted to develop and to describe variant management concepts suitable for using it in the development of safety critical systems. Starting with an analysis of variant management mechanisms in standards as AUTOSAR and EAST-ADL as well as a presentation of different variant management approaches and tools, the variant management concept for the SAFE meta model is presented. Subsequently, the usage of the mechanisms is shown in an example.

4 Introduction

Traditional product development creates multiple products overtime, often by using reuse strategies like “Clone and Own”. “Clone and Own” means new projects start by copying/branching most/all of an existing product and start their development from there. It is cheap in the beginning but eventually cost is not much lesser (or even higher) than for an approach not trying to reuse by copying. One of the reasons for this reuse “inefficiency” is that effort for activities like testing, maintenance, and certification (e.g. safety analysis, function and technical safety concept) does not reduce significantly by copying the assets, as they have to be carried out for each for the clones individually.

Especially in development processes driven by the ISO 26262 it is not possible to assemble a product by copying since the safety activities start with a hazard analysis based on the intended use. The traceability and requirements coverage from the initial hazard analysis to the various safety concepts to the final development of hardware and software components need to be fully established. To support variable design with safety constraints integrated measures are needed from the beginning. Of course, the scope of the hazard analysis could be extended to fit all possible use cases for all variants, resulting in an extremely expensive development, leading the idea of system re-use ad absurdum. If it is intended to use an item as a base-design for a product family, the possible effect of design variations to the safety activities need to be stated while performing the different verification and validation activities and safety analysis activities, to allow an exact determination of the influence in other variants.

Another often seen approach for new products is combining functionality from different versions/branches of the system in an unmanaged ad-hoc kind which sometimes takes more time than developing them from scratch (which is what developers will do after a few failed attempts to reuse by merging) leading to same drawbacks as mentioned before.

To tackle the reuse inefficiency problem, first it is needed to separate changes over time (aka versions/evolution) from combining functionality into products (aka variants) and treat version management and variant management as different disciplines from which both are necessary. Second, a *product line* approach shall be applied where common shareable assets and variability-containing adaptable assets forming the base of all products often also termed as platform. Changes on that platform in order to support a new product have to be done in a way that already existing products can be created from the same asset versions. Maintaining such flexible 150% solutions, from which a number of products each of them a 100% solution can be derived, over time, is much simpler than later trying to merge fixes across the different version of the assets used to build the individual products. For sure, some of the variants might get irrelevant over time, so caring for them is stopped eventually.

At the moment we have not defined what we understand under variability. In [3] the term software variability is specified, however, we do not just consider software but also other artifacts and as such we define variability following the definition in [3] but in a more general and broader sense: *Variability describes the ability of an artifact or of a system to be used in different contexts by changing or customizing some characteristics of it.* Those changeable characteristics are somewhere located within the artifacts and a notion that is usually used in all product line/variant management approaches is the term *variation point*. Variation points identify all places where members of a product line may differ from each other. Such difference may be the existence of certain model elements (optional or alternative artifacts) or *parameters*. A variation point describes all possible instantiations available at this point. Usually there are some conventions or even specific formal ways to express variation points. Furthermore, a variation point may have constraints, describing dependencies between instances of the variation point and to other variation points. They usually have a binding time such as compile time, link time or run time, at which the variation point instances have to be selected (the decision has to made).

To sum it up variation points describe where variations occur, what the choices are and how these variations are related to each other and very important is also that variation points in the different space (problem space and solution space) are connected to each other.

The aim of variant management capabilities in the context of safety related artifacts is to enable modeling and specifying variable elements or parts of such artifacts. That means with the proposed SAFE Meta Model Configuration Package it is possible to describe at which locations variability exists and under which conditions what variation can be applied. It is the intention of the task to go forward from system design variability to safety related variability. The influence of different configurations on the HARA shall be expressible.

5 Motivating Example – Steering System

An electric power steering system is used to demonstrate our variability approach, since different car variants (small vs. big, compact vs. luxury) usually have different incarnations of an EPS because of the varying characteristics of the cars. The steering system of a car provides the functionality of moving the car in the direction the driver desires by converting the applied rotational movement into a modification of the steering angle. Moreover, the driver gets feedback about the road surface and the current situation. In general, the following requirements and interactions with the environment hold for steering systems:

- the required operating force should be as low as possible and adapted to the current driving conditions,
- the number of rotations from one mechanical stop to the opposite mechanical stop shall be as small as possible,
- the conversion of the rotational movement to the steering angle needs to be precise and free from float,
- in case the vehicle is moving and the steering wheel is not touched by the driver, the steering wheel shall go back in the position of driving straight ahead automatically,
- the feedback from the road condition needs to be recognizable but shall not be disturbing,
- the legal regulations concerning the maximum operating force as well as the operating time shall be considered.

In particular the requirements concerning the operating force lead to the fact that in modern cars the steering action of the driver is usually supported by adding torque to the steering link and therefore reducing the effort the driver has to spend for steering the car. Within this section the example of an electric power steering, one particular form of those auxiliary systems, will be used to illustrate the developed concepts. In Figure 1, an overview on such a system is given.

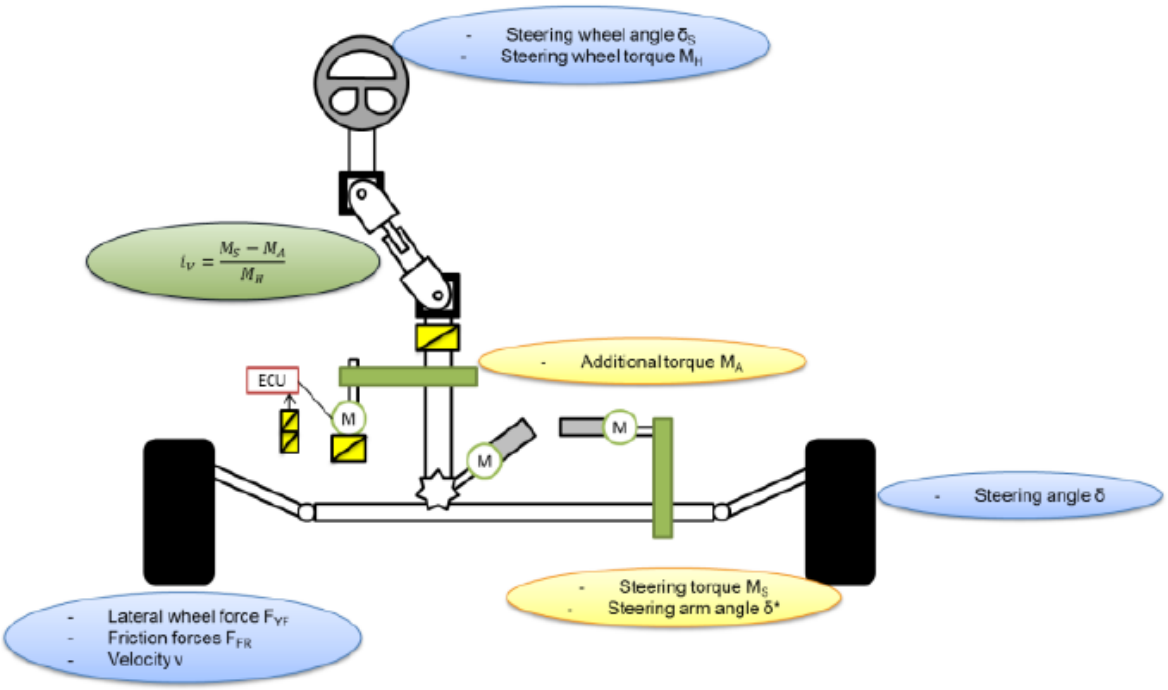

Figure 1 Overview of an electrical power steering system

In general, electric power steering systems (see Figure 1) work according to the following functional principle: The rotational movement of the steering wheel is detected by a torque sensor which passes the value to an electronic control unit (ECU). Within the ECU, the needed supporting torque which needs to be provided by an electric motor is calculated based on the received data and under consideration of other values, like, for instance, the current velocity. The electric motor is actuated by a power amplifier and passes the torque to the steering system via gears.

As a result of the item definition we obtain architecture – represented by an EAST-ADL model – that shows the item, its boundaries, and how it is embedded in the overall system consisting of the vehicle including other items and elements, the environment, and the driver and other traffic participants. The main structure of this model is depicted in Figure 2.

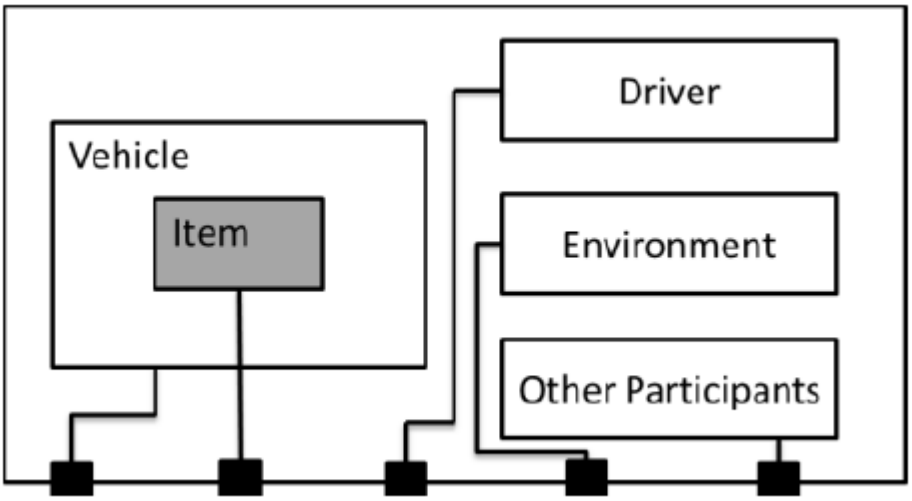

Figure 2 Electrical power steering system EAST-ADL model according to architecture definition

The hazard identified in this scenario is unintended established torque in a situation where no torque is requested:

Context: Velocity larger than 50km/h and curvature equal straight

Hazard: Within time interval of length 100ms the torque exceeds MaxTorque(x) for duration greater than or equal to x.

MaxTorque(x) expresses the maximum tolerable torque during the timeframe of 100ms.

In Figure 3 the duration for an undesired torque is depicted classified by user experiments into the categories recognizable, disturbing and uncontrollable.

This figure is drawn independently of a particular technical solution. Thinking about variable system design it is very likely that only parts of this figure might be interesting for concrete solution. Therefore the hazard and risk analysis might differ from one variant to another.

In the following sections we present an approach of how to integrate safety related properties in a variable system design.

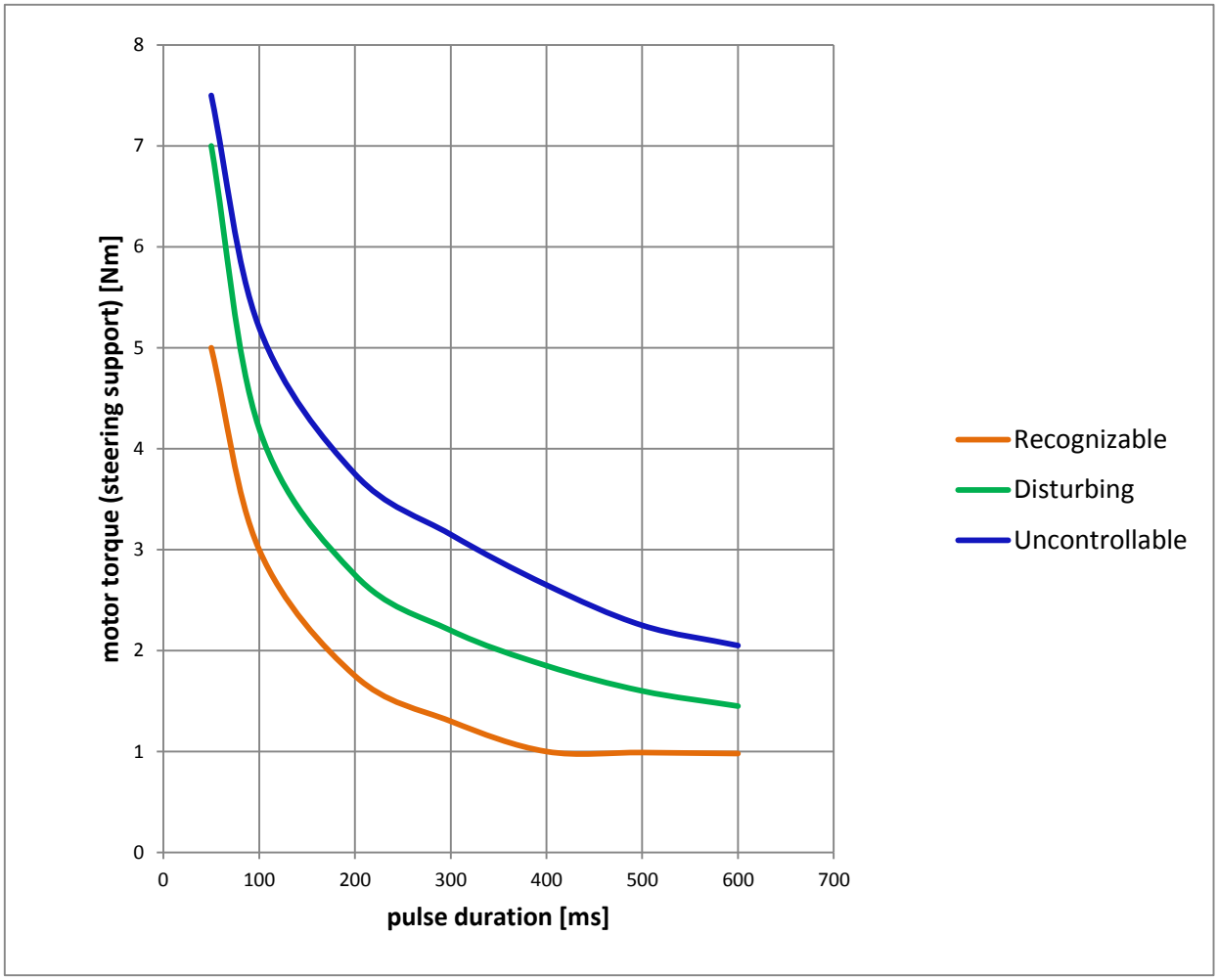

Figure 3: Relation between motor torque and time to be noticable (disturbing, uncontrollable) by the driver

6 Approaches for providing Variant Management

In this section we describe how abstract variant management information from the problem domain (e.g. variant knowledge captured and managed with feature models) can be connected with artifacts or elements that implement the solution (solution domain). In general, three different approaches are possible and depending on the given artifact language or tool the suitable has to be chosen:

- Variant Information Mapping
- Annotation
- Asset Referencing

If the used language or tool does not support at least one of the above alternatives variant management will not be possible for those artifacts. In the following we explain the three approaches in detail.

6.1 Variant Information Mapping

The variant information mapping approach is applicable for languages or tools or generally speaking for those Meta Models having some kind of support for variant handling mechanisms. That means, in the Meta Model in question it is possible to model variation points and furthermore to describe conditions that, if being evaluated, lead to one or the other instantiation realizing a specific variant at the end. The tools that implement the respective Meta Model are then able to perform the transition from such variant-rich artifacts to variant-specific artifacts through binding variation points by resolving the particular conditions and selecting the necessary parts and removing the irrelevant ones.

As an example for such language and also for a tool, in Figure 4 the C-Preprocessor (*cpp*) language is used to model a variation point in the source code by framing some part of the code with *#ifdef* and *#endif* statements. Depending on the value of the *FLAG_A* the source code between the preprocessor statements is contained in the preprocessed file that is given to the compiler for further processing. That means, to create the demanded variant the *cpp* needs to be provided with the right values for all to be evaluated *#if* statements.

Until now we just described the solution side and how the variation points are modeled and bound/resolved there. The connection to the problem space is still missing. As mentioned, to be able to create the needed variants on the solution side the values of the FLAGs have to be known by the *cpp*. That means the abstract variant management information for example features has to be linked on one side with FLAGs and on the other side with the respective values for the FLAGs. This information is usually part of a model managed by a variant management tool which might be able to generate for example a file like *defines.h* (see Figure 4) from it. Another way to provide the

```
#include “defines.h“


…
#ifdef FLAG_A == 1
// variant specific code
…
#endif
…
```

Figure 4 Modeling a variation point using the C-Preprocessor

FLAGs and their values to the *cpp* would be to give them as additional command line parameters or to create a specific *Makefile* or build script that steers the compilation as a whole. By using this approach conditional file inclusion may be realized too.

The variant information mapping approach using the C-preprocessor is applicable for all text based languages like C/C++, Java, etc. and also for all text based files in general because the *cpp* is just a simple text replacement tool. Nevertheless, the variant information mapping approach is not just useful there and consequently there exist other languages that support also variant handling mechanisms like AUTOSAR 4.0, Matlab/Simulink, where the same approach is applicable.

6.2 Annotation

If the artifact's language does not provide basic means for the description of variation points by itself another approach needs to be taken to support variability on the artifacts level. The annotation approach is being useable if the underlying artifact's meta model has either an extensible data model or provides on artifact level a free data space for tool specific data. UML goes to the first category because with so-called profiles additional elements are definable e.g. a specific variant information constraint might be specified that can be put on arbitrary model elements (see Figure 5). An example for the second category is AUTOSAR even for releases prior to 4.0 since each AUTOSAR model element has a data member called "ADMIN-DATA" where

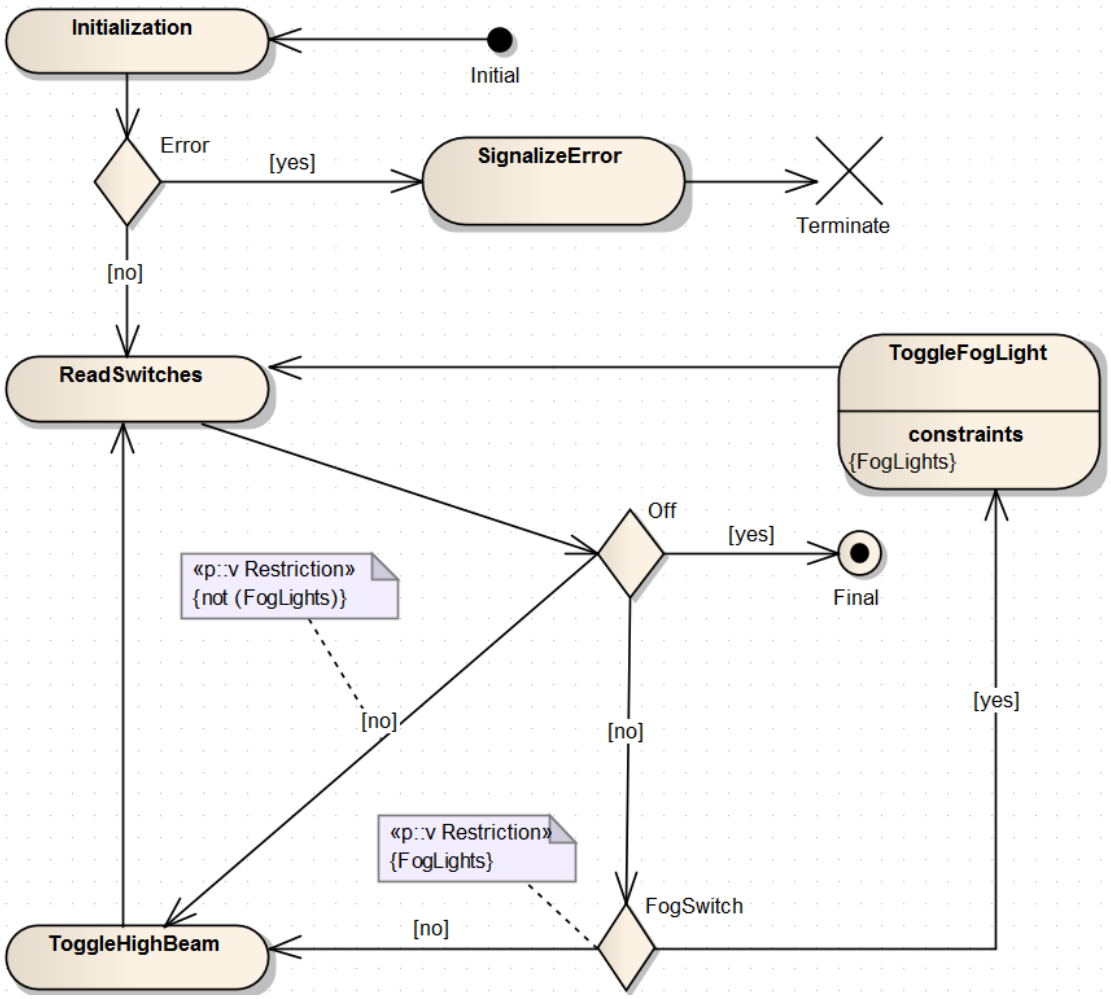

Figure 5 Variant management information (feature FogLights) is annotated via constraints on transitions and on the state ToggleFogLight

tools may store additional tool specific information in so-called special data groups (SDGs).

Depending on the respective language variant management information like features are annotated in such additional data fields denoting usually the existence condition of the artifacts. As those languages themselves do not know anything about the concept of variability at least two actions have to be carried out by the variant management to materialize specific variants:

- Resolve the variability of the enriched artifacts by binding variation points with variations
- Create variant specific artifacts by transforming the 150% model into a 100% model
    - Done by the variant management itself meaning the knowledge of the relevant parts of the respective meta model has to exist
    - Done by a specific modeling tool under guidance of the variant management

An advantage of this approach is that no additional models as for the variant information mapping approach are needed and the relevant variant information like the restrictions is stored together with the affected artifacts. Nevertheless, since the considered language and also corresponding tools are unaware of that information and as such losing this information is possible if not being worked carefully.

6.3 Asset Referencing

The asset referencing approach is the third possible approach for linking variant information from the problem domain with artifact from the solution domain. In contrast to the annotation approach the conditions e.g. for the existence of artifacts is not stored in the artifacts itself within the artifacts model but instead in an additional model called the referencing model (see Figure 6). This is usually due to the fact that the solution domain model does not support embedding additional information on artifacts, meaning the data model is not extensible.

The statements according to the actions a variant management suite has to perform are the same as for the annotation approach since both are very similar from the variant management perspective except the fact where the variant information e.g. conditions are stored. The downside in contrast to the annotation approach is that an additional model is necessary that has its own life

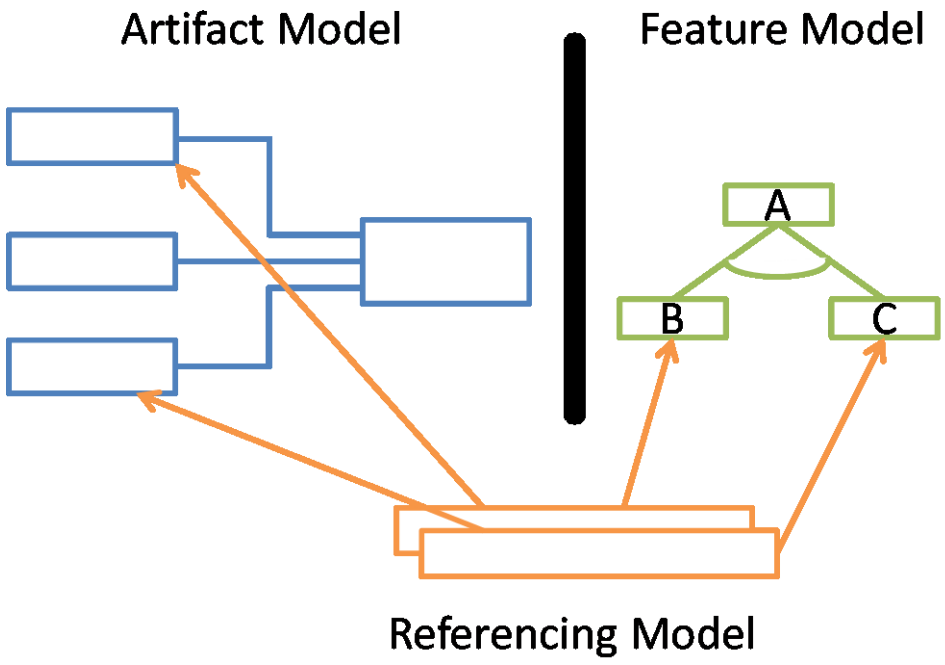

Figure 6 Artifacts are linked with variant management information via a referencing model.

cycle and evolution speed, meaning one has to ensure that the information are in line or synchronized between the related models.

An example where the referencing approach is used for the variant management is EAST-ADL. A second example is pure::variants with its generic variant management support for EMF ecore models where the low level identifiers of ecore model element representations are liked with features to enable variability modeling for arbitrary EMF artifacts.

7 Variability Mechanisms in existing Standards

In the following section we will discuss the variability mechanisms that are provided by existing standards. We show what is supported each time and how it works. As relevant standards for the automotive industry and as such for the SAFE project, we concentrate on AUTOSAR as well as on EAST-ADL.

7.1 AUTOSAR

With the advent of AUTOSAR in 2002 and with its first release in 2005, it became the de-facto standard for the automotive electric/electronic architecture and system design. Since that time, the standard has further developed and made major progress. Currently, the latest AUTOSAR release is release number 4.0 from 2009 with some later maintenance releases. However until today, the industry does not fully switched to that release and thus former releases are often applied, mostly having a 3 as major release number.

Considering the challenge of variability, we have to answer the following question:

- Does the high-level abstraction of AUTOSAR supports modeling of variability. If not, are there any other ways to enable support for variability, in general?

Answering the question is not a clear yes because it depends on the AUTOSAR release, whether support for modeling variability exists or not. All releases prior to AUTOSAR 4.0 do not know a notion of variability but since release 4.0, variability may be expressed [1]. Due to still strong adherence to AUTOSAR release 3.x of some automotive OEM the question arises: Can variability be supported in those releases? Likely, because the AUTOSAR meta model permits providing additional information on model elements by using so called admin-data. This way, tool specific data, for example for modeling variability, can be integrated using the annotation approach. Another way could be using the asset referencing approach.

One remains still to answer regarding variability modeling in AUTOSAR release 4.0, is variant handling or also variant management supported? The AUTOSAR meta model supports just variant handling, meaning to express that a model element is subject to variability. However, it is not possible to specify or to define dependencies between thus labeled elements. Variant management was not in the scope of the development of the AUTOSAR release 4.0. That implies variant management has to be carried out without AUTOSAR meta model support. Usually, external tools for variant management are used like pure::variants [2].

The following description of variant handling mechanisms applies only for AUTOSAR release 4.0 and later releases.

7.1.1 Variant Handling

- Only Variant Handling no Management
- Just selection and parameterization
  - Existence of model elements (*aggregation, association*)
  - Parameterization (*attribute value, property set*)
  - Relations between elements according to variability cannot be modeled
- Where:
  - Not every element is subject to variation
  - Meta Model defines locations by <<*atpVariation*>> annotations
- When:

- dependent on variability kind (aggregation, association, …)
- All relevant automotive binding times (from system design till to start up time)

7.1.2 Variantion Point

- Variation Point mechanism
  - just for structural variability
  - *PreBuild*
    - Different binding times
    - Condition based on *swSystemconst*
    - Single condition
  - *PostBuild*
    - Explicitly not run-time but start-up time
    - comparison of values and their *PostBuildVariantCriterium*
  - *PreBuild* and *PostBuild*, even together
- Modeling is not uniform
  - E.g. Attribute values are treated differently

7.1.3 Variant Definition

- Based on *SwSystemConst*
- Hierarchical definition
  - 0-n *EvaluatedVariantsets*
    - 0-n *PredefinedVariant*s
      - 0-n *SwSystemconstValueSet*s
        - » 0-n *SwSystemconstValue*s
- Potential problems
  - Non disjoint sets on any hierarchical level
    - Needs to be ensured by tools
    - Merging difficult due to possible conflicting values
    - Problem detectable by means of model checking
    - Stated in the standard as additional constraint that tools have to be fulfilled
  - Referencing system constants that itself are subject to variation

7.2 EAST-ADL

- Variant Handling and Management
- Variant Handling
  - Basis Concept: Variation Point
  - VariableElement and VariationGroup

  - Each element may be subject to variation
- Variant Management
  - Vehicle Feature Modeling
  - (hierarchical) Feature Modeling
  - DecisionModel
- Problem:
  - No element(s) for representing a variant – Question: How is a variant defined?

8 Variability Mechanisms in existing tools

8.1 Variant management in PREEvision v5.5

PREEvision provides concepts for product line and variant management [3]. Variant management in the tool environment facilitates to handle different products in a *Product Line*, as shown in Figure 7. A *Product Line* of an electric and electronic architecture is modeled as so called 150%-model. This model includes all information about *Product A*, *Product B* and *Product C* in a single consistent model. Using the integrated variant management the different products can be tailored into different subsets.

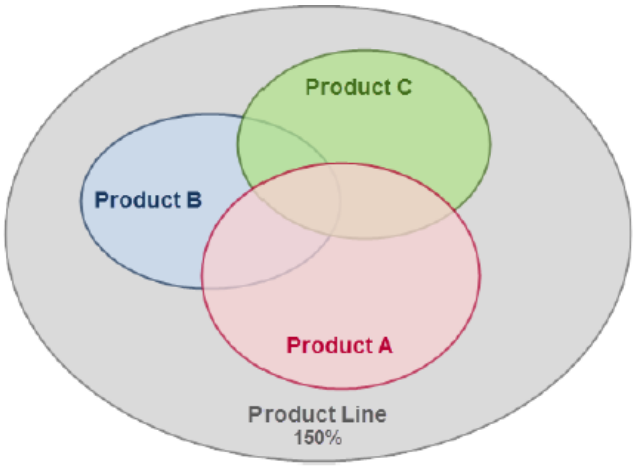

Figure 7: Variant management [3]

8.1.1 Structure of the variant management

The structure of the variant management is shown in Figure 8. Artifacts of all abstractions layers can be used for the variant management. The structure of the variant model provides a ***Concept Space***, which includes ***Architecture Variants***, ***Concept Templates*** and *Alternatives*. Additionally, ***Equipment Templates*** including *Alternatives* and ***Sets*** are available. These are described in detail in the following.

***Concept Space*:**

The *Concept Space* is a kind of design space to bundle possible variants of architectures. A *Concept Space* can represent a vehicle with all points of variants. It represents the basis to create architecture variants.

The *Concept Template* describes all possible implementations of a certain assembly of the vehicle and is directly associated with a *Concept Space*. This constitutes a technical partial solution. The *Concept Template* holds the assigned *Alternatives* and let the user specify exactly one *Alternative* for one *Concept Template*.

An *Architecture Variant* is one specific variant related to exactly one *Equipment Template Alternative*.and can be described by one or more technical implementation concepts. The sum of the chosen *Concept*- and *Equipment Template Alternatives* results in one concept implementation variant of the according *Concept Space*. It is possible to create various *Architecture Variants* to structure variants in a well arranged way.

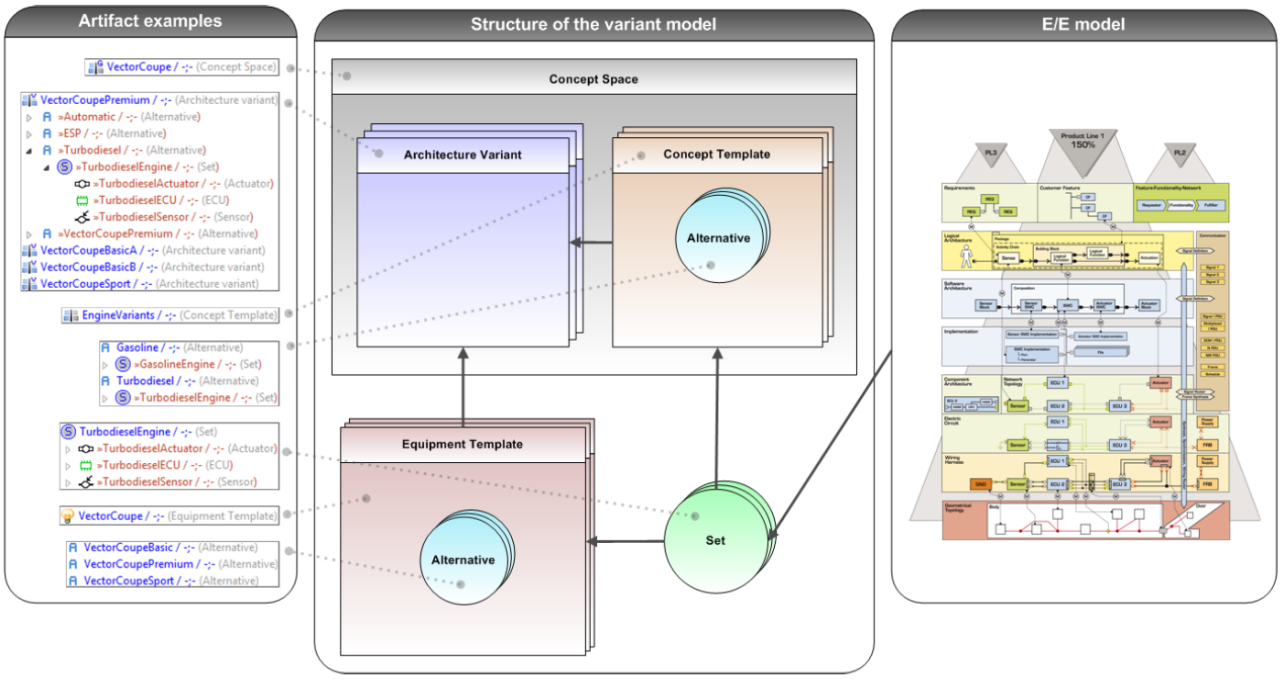

Figure 8: Variant management structure in PREEvision [3]

***Equipment Template*:**

The *Equipment Template* describes equipment characteristics from the customer point of view. It holds *Alternatives* and let the user chose exactly one of them for one *Architecture Variant*.

***Set*:**

Sets are basic structure elements and artifacts of the abstraction layers can be grouped. They are highly reusable and therefore a good granularity has to be found in order to use them in multiple templates [3].

***Alternative*:**

Alternatives can be used in *Concept Templates* and *Equipment Templates*. The *Concept Template Alternative* represents one specific technical solution for a *Concept Template*. The *Equipment Template Alternative* represents the available vehicle configurations. Exactly one Alternative can be specified to a Template.

8.1.2 Utilization of variant management in PREEvision

8.1.2.1 Configuration with variant perspective view in PREEvision

Figure 9 shows the variant management using the corresponding variant management perspective of PREEvision ①. The selection of an active variant using the scrollbar ② facilitates that all assigned artifacts will be shown in the model view ⑤, all other artifacts will be faded out. Using the variant management perspective the user can create or configure architecture variants of the different concept spaces ③. In addition the user can edit Sets or assign them to Templates ④.

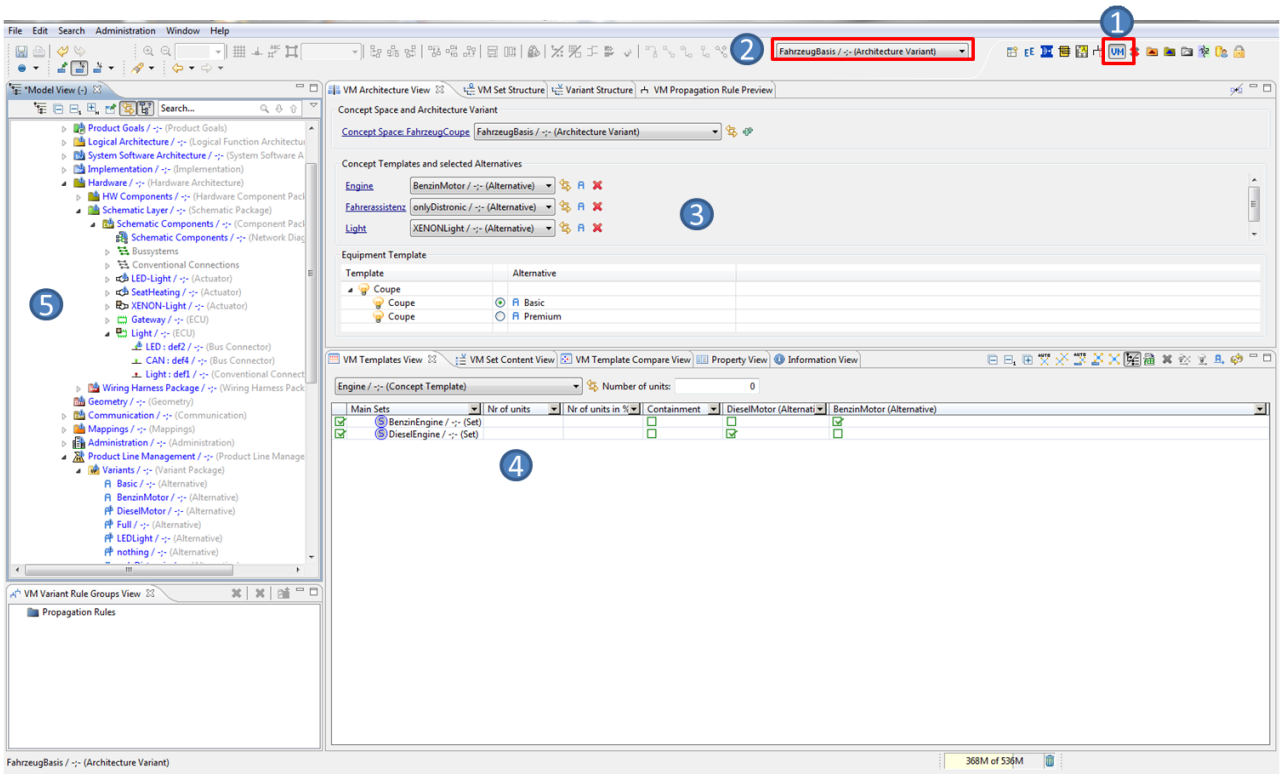

Figure 9: Variant perspective

8.1.2.2 Example for usage

Figure 10 shows a simple example of a front-light system. There are two possibilities: “LED-Light” and “XENON-Light”. The Sets (1 and 2) include the artifacts of the different light-systems. These two Alternatives are assigned to the *Concept Template* named “Light”. Therefore the Architecture Variant “VehicleComfort”, shown in Figure 10 with red frame, can be varied according to the different *Alternatives* of the *Concept Template*.

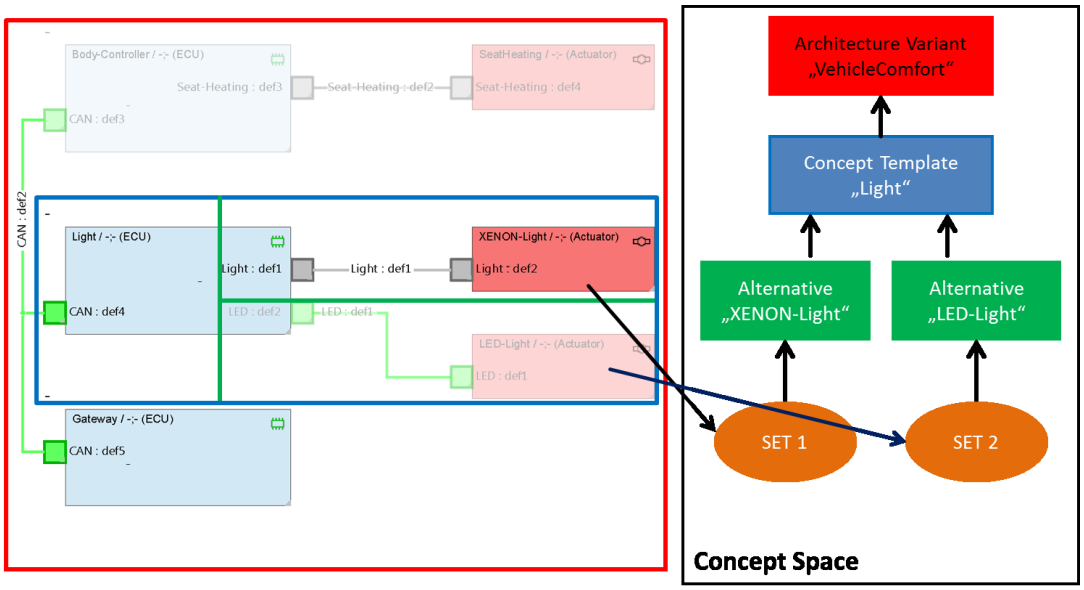

Figure 10: Example for using variant management in PREEvision

8.1.2.3 Additional use cases

Additional use cases for the utilization of the variant management is the analysis of an architecture regarding exploration of the design space and optimization, comparison of different architecture realization alternatives and for simple grouping of artifacts to make them easily accessible.

8.2 pure::variants

pure::variants is a tool for efficient variant management supporting the whole life cycle of product lines starting from the requirements phase through design, architecture, and realization phase up to the testing phase. Due to its independence for example of programming languages or modeling tools it integrates nearly seamlessly into existing development processes.

The basic idea of pure::variants is the separation of concerns and as such the distinction between a problem domain that considers the variability of the system from an abstract point of view and a solution domain that deals with implementation artifacts like requirements, architecture models, source code, etc. pure::variants captures the problem space within feature models and the solutions with family models separately and independently in a structured way (see Figure 11). Additionally, those models allow defining inter-relations and restrictions on the building elements that cannot be expressed by just structuring the information in the right manner. From the management perspective the models allow a uniform representation of all variabilities and commonalities of the products of a product line and the therein captured knowledge provides the basis to perform informed, comprehensible and valid decisions during the definition of variants.

In contrast to other feature model based approaches which merely allow the use of only a single feature model, pure::variants facilitates the use of multiple feature models that may be hierarchically linked. This approach ensures feature combination consistency and enables and simplifies the re-combination of product lines or parts of them into new ones.

Regarding development, pure::variants supports the recording and management of all models and information during all stages of the development independent whether it is targeted to problem space analysis or solution design. Furthermore, it can assist for example by obtaining a software family implementing a product line by providing means that enable the transition from a set of legacy software implementations realizing similar products.

For the definition of a variant out of all these models, a user (developer or customer) selects just features required for the desired variant or product from the feature models. pure::variants checks the validity of this selection and if necessary automatically resolves dependency conflicts. Thereby it maintains the integrity of dependencies, no matter how complex they are. Having a valid selection is the trigger for an evaluation of the family models that contain the solution definitions consisting of requirements; models source code, logical and physical parts, etc. The evaluation process results in an abstract description of the selected solution in terms of artifacts. This description controls the transformation process that creates for example the needed software solutions.

At every stage of the evaluation process conflicts are solved automatically or, should this not be feasible, are handed over to the developer for solving.

8.2.1 Product Line Development

As denoted the enterprise tool suite pure::variants is usually used in different phases of the software development process. Such process or better the development of a product line is basically divided into two steps in general. In the first step the problem and solution domains are analyzed, common assets are identified and realized (domain engineering). In the second step the individual products are derived from the product line (application engineering).

Several model types are used to capture the information required to manage variability and variants on the different levels of domain knowledge, software design and implementation (see Figure 11). *Feature models* play a key role in this. They allow a uniform representation of variabilities and commonalities of the products of the entire product line. Implementations of the product line are described by family models. They enable the mapping of the problem space to the different implementations. This model type was developed especially for the pure::variants technology, since existing modeling techniques such as UML or SDL were not suitable for this

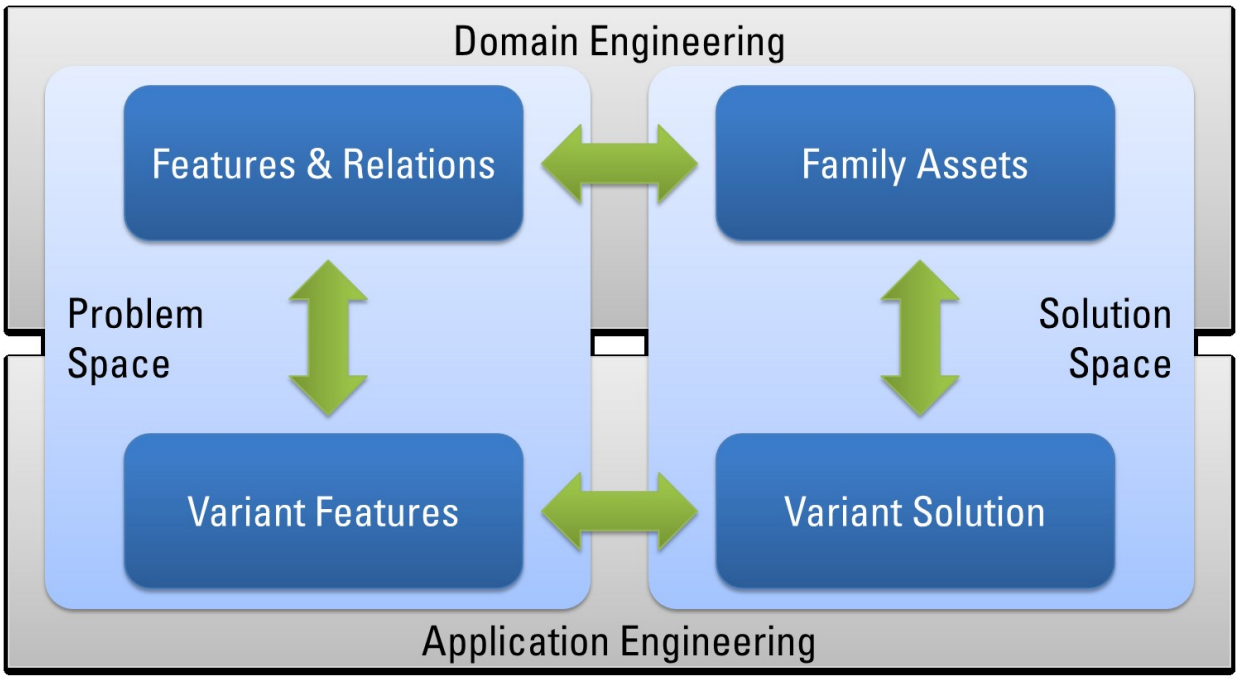

Figure 11 pure::variants' terminology

purpose. The variant description model is used to describe an individual product. It describes the product's features and values associated with those features, and it is used to derive the final product from the family models.

8.2.2 Domain Engineering

The principle operational sequence of such a development process is described in the following, beginning with the identification of the common assets.

8.2.2.1 Analysis of the problems:

Based on a content-wise problem analysis, feature models are built to capture the dependencies between the individual features of the product line. Feature models are easily visualized and conceived.

Based on the experience with the practical use of feature models, the expressiveness of pure::variants's feature models was substantially extended. The support of model hierarchies in particular enables different representations of the problems depending on the user (customer, developer, sales, etc.).

8.2.2.2 Design of the solutions:

Starting from the feature model and in combination with further requirements for the software systems, the design of the software solution is performed. The elements of the software solution with their relations, restrictions and requirements are integrated into the family model and hence are available for automatic processing. Family models might be divided into several levels. For the further description, we use an example of a software product line and we describe how potential family models could look like. On the highest level so-called components are used and each of the components represent one or more functional features of the solutions and consists of logical parts of the software (classes, objects, functions, variables, documentation). In the next level, the

physical elements of the solution are assigned to the logical elements. The physical elements can be files that already exist as well as files that are to be created and actions that are to be performed based on the variant knowledge.

Since pure::variants captures the problems (feature model) and the solutions (family model) separately and independently, the re-use of the solutions and of the feature models in new projects is simplified.

8.2.2.3 Realization of the solutions:

The solutions are realized by employing the capabilities of the selected and used programming language and tools, and by using the additional possibilities of generating variants provided by the pure::variants transformation modules. Not only the creation of new product lines but also the integration of existing software (re-engineering) into a product line based development in particular is easily possible with pure::variants. For this purpose the product line development process does not start with the identification of the common assets but with the (partial) automatic production of the family model based on the already existing software and with the step-by-step construction of the associated feature model by the users. The remaining steps are similar to the process outlined above.

8.2.3 Application Engineering

Creating variants with pure::variants is relatively straight forward once the different models are produced because the remaining steps may be performed automatically. The developers of the product line are responsible for providing the feature model as well as the solution description with family models. The user then selects the features. Here, the user can be either a human or a tool that determines the necessary features automatically based on the application. Further processing is performed in two steps. At first pure::variants analyzes the different models. The result is a construction plan from which the customized component of the final product is derived in a second step, called transformation. The transformation step is configurable by the product line developers/users.

8.2.3.1 Determining a valid combination of features:

The user (customer, sales, and application developer) selects the features relevant to the problem solution from the feature model. pure::variants checks whether the feature selection is valid and, if necessary, resolves dependency conflicts automatically. This ensures that even complex dependency structures can be efficiently transformed into a valid system. The result is a combination of features that describes the problem to solve as intended by the developers of the models.

8.2.3.2 Selecting a suitable solution:

Based on the selected features, the family model is used to find a suitable solution. Using the information contained in the family model, the feature selection is analyzed by each component and its logical and physical elements. For each part it is decided whether and in which form it belongs to the solution. Problems that may arise due to further dependencies are resolved automatically if possible or handed over to the user for manual solution. The strategy for the selection of the best suitable solution is in general manually realized by the users. However, it is possible to integrate problem-specific strategies into pure::variants if necessary, depending on user demands. Thus it is possible to select a solution according to optimization parameters, e.g. according to the customer's cost model for parts of the component. The result is a description of the selected solution in form of a component description.

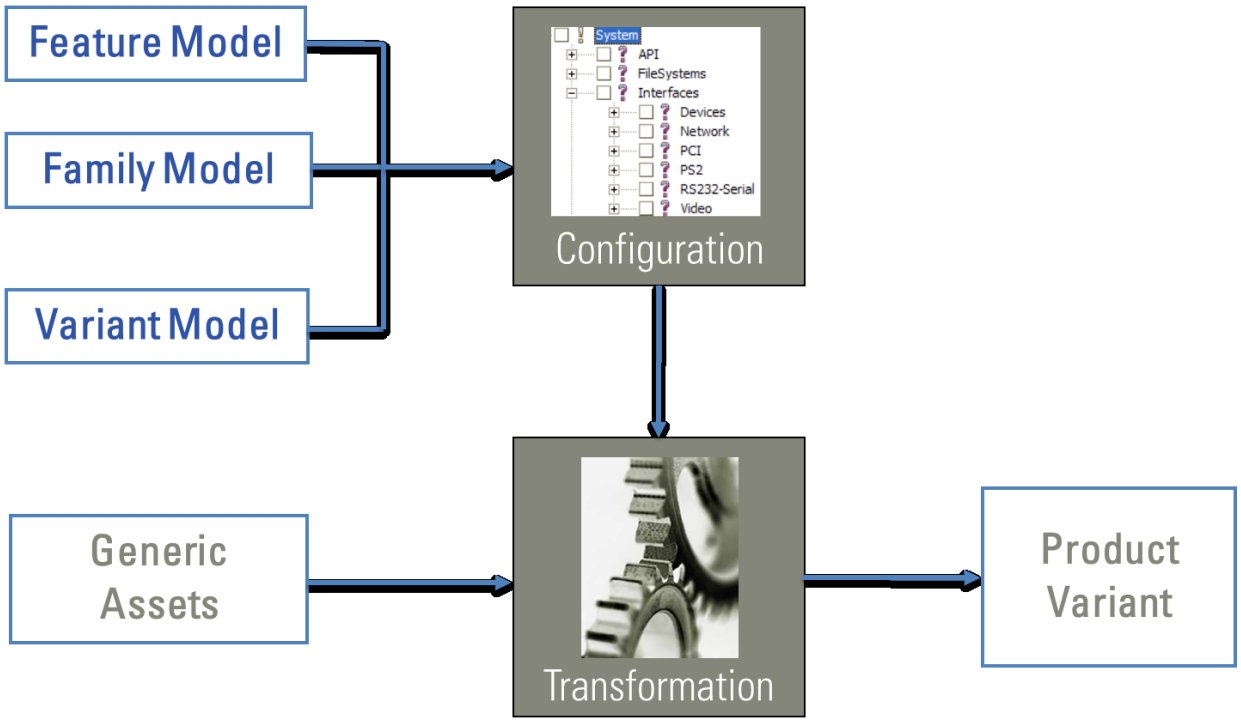

Figure 12 Relations and data flow during transformation

8.2.3.3 Creating the solution:

The customized software solution is created by a transformation process controlled by the different used input models (see Figure 12). During this process all necessary transformation modules are activated and perform the conversions specified in the component description, resulting in a variant-specific solution.

9 SAFE Variability Modeling Approach

This section describes the variability modeling approach we have taken within SAFE for the first iteration. We show the SAFE Meta Model Configuration package and explain how the modeled meta classes work in general.

9.1 General Modeling Approach

For the integration of variant management information with safety related assets, we chose the Asset Referencing Approach. However, that is not a pure Asset Referencing Approach in a general sense, where the relation between variant management information (e.g. features) and SAFE model elements is held or modeled outside in an additional model. Instead that information is also contained within a SAFE model to adhere with the single source of information principle but extracting the information and storing it into a separate model is easily doable. Nevertheless, that was not the reason for choosing the asset referencing approach because looking at the considered standards AUTOSAR and EAST-ADL and especially at their meta models and there to the top-level structure with the abstract base meta classes one will recognize that both have something in common. All relevant model elements are identifiable, which is a prerequisite for the Asset Referencing Approach. Reasoned by this fact, we modeled the SAFE top level structure accordingly. That means further we can use the mechanisms realized in WT3.4 and implemented in WT4.2.5 in the same manner for SAFE as well as AUTOSAR and EAST-ADL. Although both standards have own variability mechanisms driving them from a common base as it is our proposal could be very beneficial and convincing and we reach in this way a consistent variant management through all related models.

9.2 SAFE Meta Model Configuration Package

The SAFE Meta Model Configuration Package presented in Figure 13 contains just three meta classes (*Restrictable, Restriction and Variant*) building the minimal set to allow tagging artifacts as being variable and to specify Variants. The structure of the package, the classes and the relations between them are visualized and furthermore the class *Identifiable*, which is a class form the top-level structure and a prerequisite for the Asset Referencing Approach

9.2.1 Restriction

Generalization:

- none

Description:

A Restriction is used to decide if an element is part of a variant's resulting configuration.

Attributes:

- expression : String = true [1]
  The string specifies a condition (e.g. a formula expression consisting of features) that can be evaluated to *true* or *false*. An example for such expression would be "FeatureA and FeatureB".

Associations:

No additional associations

Constraints:

No additional constraints

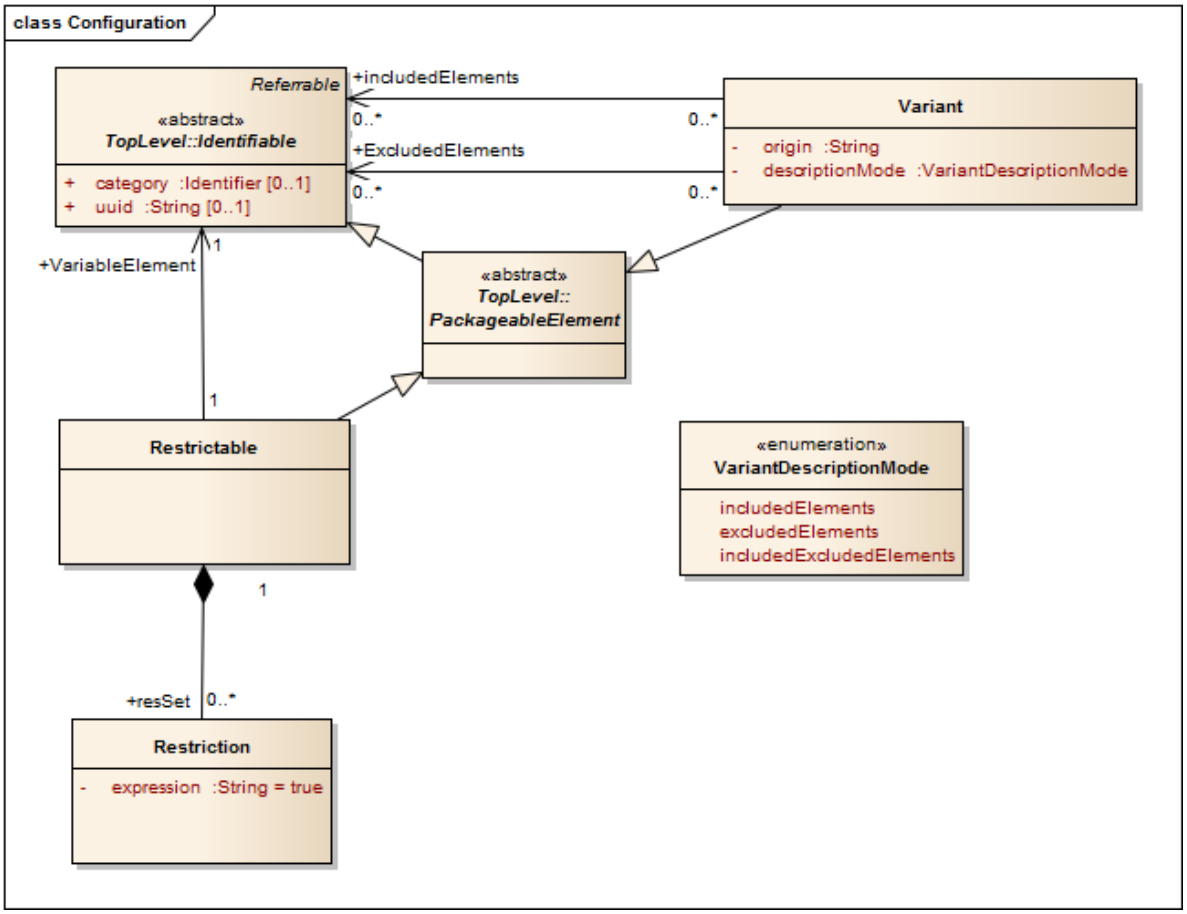

Figure 13 Safe Meta Model Configuration Package

Semantics:

A Restriction defines the present condition of an artifact.

9.2.2 Restrictable

Generalization:

- PackageableElement (from TopLevel)

Description:

A Restrictable allows connecting SAFE model elements identified by the association VariableElement with variant information. The identified element is denoted as an optional element that is not part of all variants. The decision of being part of a variant is defined by the set of restrictions. If at least one restriction evaluates to true, the element will be contained.

The Restrictable itself can be part of an arbitrary package, however, this may change in future iterations.

Attributes:

No additional attributes contained.

Associations:

- VariableElement : Identifiable [1]
  The association points to the element in the SAFE model that is variable.

- resSet : Restriction [0 ..*]
  The association holds the set of restrictions that constrain the availability of the element identified by VariableElement.

Constraints:

1. The VariableElement may not identify elements within the configuration package.
2. If resSet is empty, no restriction is given and the identified element is always part of variant.

Semantics:

The semantic of the multiple restrictions within the reset is “or”.

9.2.3 Variant

Generalization:

- PackageableElement (from TopLevel)

Description:

The class *Variant* is used to reference those model elements of a 150% SAFE model that are relevant (includedElements) and/or irrelevant (excludedElements) for a specific variant. Since *Variant* is derived from *Identifiable,* the *shortname* attribute can be used to give the variant a specific name.

Attributes:

- origin : String [1]
  This attribute holds information (e.g. configuration management information) in order to allow checking if the *Variant* is up-to-date to the actual referenced model elements.

- descriptionMode : VariantDescriptionMode [1]
  The *descriptionMode* describes the instantiation mode of the *Variant.* A *Variant* can be instantiated using one of the associations includedElements or excludedElements only or using both together. The enumeration *VariantDescriptionMode* specifies those three modes.

Associations:

- includedElements : Identifiable [0 .. *]
  The association points to elements in the SAFE model, that are relevant/included in the *Variant*.
- excludedElements : Identifiable [0 .. *]
  The association points to elements in the SAFE model that are irrelevant/included in the *Variant*.

Constraints:

1. The includedElements may not identify elements within the configuration package.
2. The excludedElements may not identify elements within the configuration package.

Semantics:

9.2.4 VariantDescriptionMode

Generalization:

- none

Description:

The class is an enumeration, which defines three different modes how a variant can be materialized.

Attributes:

No additional attributes contained.

Associations:

No additional associations contained.

Constraints:

No additional constraints.

Semantics:

No additional semantics.

9.3 Limitations

The meta model extension regarding variability supports the labeling of optional elements, meaning the selection of optional additional model elements for certain variants and furthermore it supports defining variants. That is basic functionality only but not a problem in reality because with it one can realize the most common cases. Nevertheless there is some functionality, that would be useful, but not supported by the SAFE meta model:

- Expressing and modeling the relation of arbitrary artifacts like requires, conflicts, etc.
- Support for variable parameters that depend for example on feature attributes

10 Application of Variability mechanisms to steering system

While performing the Hazard and Risk Analysis (HARA) of the Electronic Power Steering System (see Section 5) the unintended creation of torque while driving on a straight road has been identified as a hazardous event:

Context: Velocity larger than 50km/h and curvature equal straight

Hazard: Within time interval of length 100ms the torque exceeds MaxTorque(x) for duration greater than or equal to x.

The hazardous event now needs to be classified to determine the ASIL. We want to focus on the controllability and assume the Exposure is E4 and Severity is S3:

Class	*Level*	*Rational*
Exposure	E4 (High Probability)	Driving above 50km/h on a straight road is a very common driving situation.
Severity	S3 (Life-threatening injuries (survival uncertain), fatal injuries	Getting off the road unexpectedly may lead to serious harm.

Nevertheless the controllability is not easy to determine, since it depends upon the force and duration that is expected to occur. Figure 14 depicts the relation between the applied torques, the duration of the torque and the Controllability level.

The classification is highly relevant for the resulting ASIL of the hazardous event, and therefore has a major influence on the costs of the item. Table 1 lists the resulting ASIL in dependency of the controllability level:

Controllability	*ASIL*
C0	QM
C1	B
C2	C
C3	D

Table 1: Relation between Controllability and ASIL of the Hazardous Event

Without knowing anything about the technical solution a controllability of C3 had to be assumed, resulting in an ASIL D system. However, depending on the configuration of a variant the system might be equipped with a motor that is not capable of creating such high torque to reach the C3 region (see dashed line in Figure 14). That means, the system has to be developed to ASIL D anyway or according to the variant's configuration just with a lower ASIL. Going this direction implies an alignment of system architecture and related safety analyses since both depend on the actual variant.

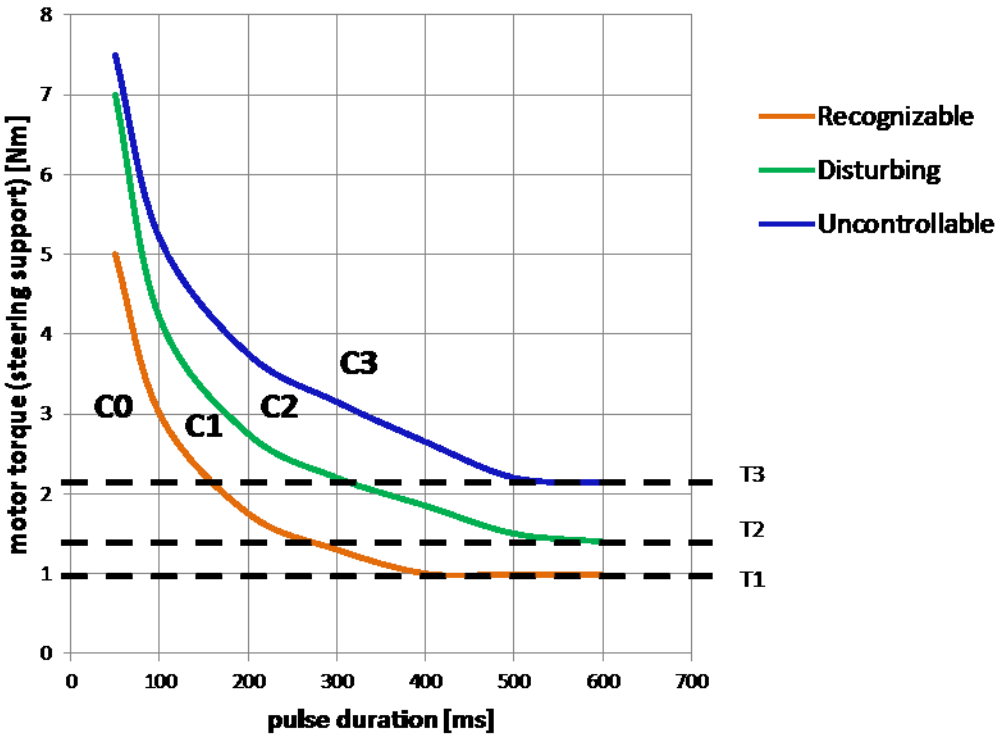

Figure 14: Classification of the Controllability

If we do not do the alignment, the re-validation and re-verification of the item analysis and HARA will be unavoidable and therewith the use of variability points in the classification of the hazardous event is suggested.

In Figure 15 restrictions are used to allow variability in the controllability level of a hazardous event. The torque levels T1 – T3 are highlighted also in Figure 3. The MotorMaxTorque (MMT) is available by the design variability model. In this case it is assumed that the motor in the electrical power steering is outside of the item and can be purchased separately.

This kind of modeling has a strong influence on the V&V activities in the process. Without variable handling of hazardous events either the resulting ASIL is too high or the impact of the change for the HARA, the Functional Safety Concepts and also the Technical Safety Concept has to be determined which is a major activity. On the other hand it is much easier to integrate the variable points while the concepts are being created.

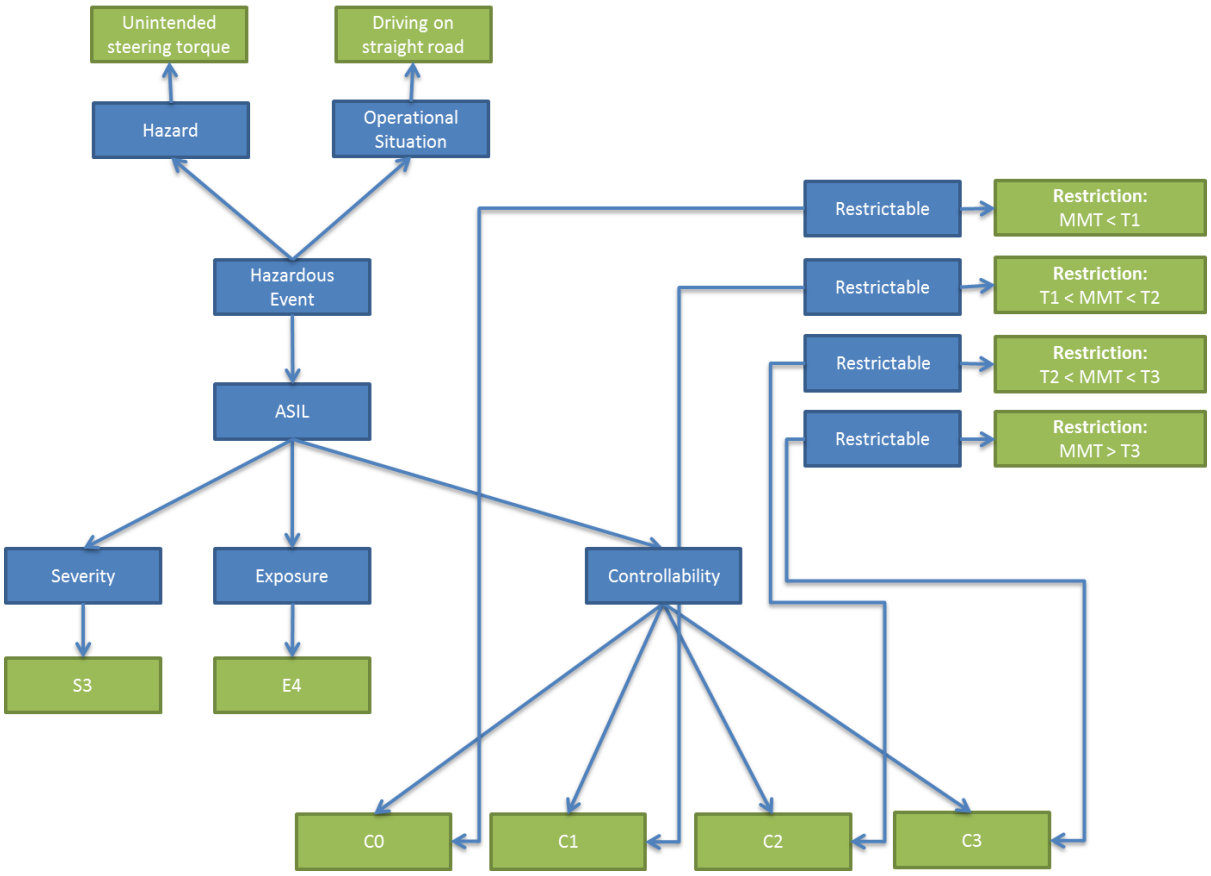

Figure 15 Hazard analysis containing variability

11 References

[1] AUTOSAR: Generic Structure Template.

[2] pure-systems GmbH: *pure::variants Eclipse Plugin User Guide*, 2011.

[3] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *2nd Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.

[4] PREEvision v5.5.2 User Manual, Vector Informatik GmbH ISO 26262, 2012

12 Acknowledgments

This document is based on the SAFE and SAFE-E projects. SAFE is in the framework of the ITEA2, EUREKA cluster program Σ! 3674. The work has been funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS11019, and by the French Ministry of the Economy and Finance (DGCIS). SAFE-E is part of the Eurostars program, which is powered by EUREKA and the European Community. The work has been funded by the German Ministry of Education and Research (BMBF) and the Austrian research association (FFG) under the funding ID E!6095. The responsibility for the content rests with the authors.