Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines: Specification and Analysis Methods*

Stefan Mann
Fraunhofer Institute for Software and Systems Engineering ISST
Steinplatz 2, 10623 Berlin, Germany
stefan.mann@isst.fraunhofer.de

Georg Rock
PROSTEP IMP GmbH
Dolivostr. 11, 64293 Darmstadt, Germany
georg.rock@prostep.com

January 12, 2009

Abstract

Architectural description languages (ADLs) are essential means for a system and software design in the large. Their common concepts are components, ports, interfaces and connectors. Some of them already support the representation and management of variance, a prerequisite to support product line engineering, but the support of variability often stops on component level. In this paper, a more detailed view on the integration of variability into architectural models concerning the formal analysis of variability and the related assessment of the defined product lines is taken. The focus is set on the support of product line engineering within the automotive E/E domain, where functionality or its realization is varying according to specific customer needs and hardware topologies. In general, the fundamental question in this domain is not, if a product line approach is applied, but what is the best way to integrate it.

1 Introduction

Architecture description languages (ADLs) are widely used to specify systems and software designs in the large. According to nowadays complexity of embedded software systems in the automotive industry (more than 2000 functions in a today’s upper class vehicle possibly interacting with each other) architecture specifications have to be structured in different layers. At every layer we propose to use an appropriate specification formalism that gives us on the one hand the possibility to analyze the described models at each layer and on the other hand the possibility not to handle them as isolated development artifacts but to integrate them within a common development process. The layers introduce different perspectives onto a system. For example, a function architecture perspective describes a system as a net of communicating hierarchically structured functions, thus providing the possibility to introduce abstraction layers together with its logical interfaces. The function architecture incorporates a refined requirements description for the design of a software architecture, i.e. it represents a suitable basis for the design of an AUTOSAR software components architecture. During the development of complex systems and in a distributed engineering task such a structured and modular approach is indispensable to cope with the unavoidable complexity.

*This work was partially funded by the Federal Ministry of Education and Research of Germany in the framework of the VEIA project (German acronym for “distributed engineering and integration of automotive product lines”) under grant: 01ISF15A. The responsibility for this article lies with the authors. For further information cf. the project’s website: http://veia.isst.fraunhofer.de/.
In the recent past software product line based development methods truly became a mainstream development technique. Although successfully applied in the software development field, product lines are still not state of the art within the automotive domain of embedded software systems, although the vehicles’ software systems are de facto product lines. Thus, the software and E/E system product lines are built and managed often explicitly by naming and configuring all variants in advance and maintaining all single products through the complete development process. There is an insufficient support of a pervasive and consistent variability management embracing all development artifacts (functions, software and hardware architectures) which constitutes a prerequisite for managing product variants and their complexity.

Within the BMBF funded research project VEIA [8] we developed a concept to support the engineering of automotive system families on the basis of common product line approaches. The ability to describe variability on different architectural perspectives (cf. Figure 1) allows us to analyze dependencies between variants of functions or software components and to evaluate alternative solution strategies to correlate the (desired) variability with development costs and efforts. A prototypical implementation of a case tool we named v.control demonstrates different aspects of our methodology: the specification of product lines in form of function and software component architectures including abstraction and reuse issues, the assessment of a product line by the evaluation of the architecture specification using metrics, and the configuration of a product line architecture in order to consistently derive the specifications of each of its products. For the latter we use common feature modeling techniques for a centralized and continuous variability management.

The presented work focuses on the connection of abstract description techniques for variability with more concrete architecture layers exploiting a refinement relationships between the function and software layers. The methodology represents a prerequisite for the handling of variability at more abstract layers where the following software development is executed according to the AUTOSAR methodology [2]. Although the concepts for the handling of variability are not yet stable within the AUTOSAR considerations, this work shows a possibility to realize a pervasive handling of variability throughout a product development process that starts with the requirements specification and ends with an AUTOSAR compliant software development. As proof of concept the work is prototypically implemented in a tool that was used to develop a product line for a “condition-based service” application in the automotive area. The tool combines state of the art formal analysis techniques with the possibility to compare different development strategies according to established metrics.

In this paper we introduce the underlying concepts for the integration of variability modeling...
aspects into architectural models (see section 2) and a possibility to automatically compute the corresponding feature tree and formally analyze it (see section 4). Doing so we are able to analyze the variance of the product line with respect to architectural views and abstraction levels, to assess alternative solution strategies with respect to their variability aspects, and to evaluate them regarding development costs and efforts (section 5). We examined the presented method with the help of an example use case sketched in section 3. The paper ends by referring to related work and giving some concluding remarks.

2 Variability concepts in architecture descriptions

Automotive product lines have to be engineered within a larger system context of hardware and software systems. The proposed layered approach respects the current development artifacts and processes and introduces variability concepts only where needed. As a reference, the artifacts depicted in Figure 1 were considered within the VEIA project.

The commonality and variability between products are expressed using feature models represented as (hierarchically structured) feature trees as they were introduced in [11, 5]. Mandatory features represent the common parts of the products in a product line. Variable features represent the differences between products using optionality, alternatives (encapsulated by XOR variation points) and OR variability, cf. e.g. [11, 16, 18]. Parameterization of attributes is also often provided. Furthermore, dependencies (e.g. “needs”, “requires” or other logical statements) are used to constrain possible combinations of variable elements. Typically, the description of a single product corresponds to a complete configuration of the feature tree, i.e. for all variable features it is determined whether the feature is selected or deselected.

Logical architectures, software architectures and technical architectures are described using an architecture description language that allows to define variability of different specification elements. The logical architecture specifies the required functionality. It is directly connected via a refinement relationship to the specification of software architectures representing an implementation of the logical architecture. This relationship is used to trace changes made on different levels and to semantically connect the descriptions on these levels to allow for a simultaneous configuration process. Both the logical architecture and the software architecture are connected to an infrastructure layer via a deployment mapping. This infrastructure layer specifies the topology and hardware restrictions the other layers rely on.

The integration of variability into the VEIA architecture models is based on the fundamental concepts of common ADLs, see e.g. [14]: Components are the building blocks of an architecture. Interface descriptions (ports) specify the ability of combining components. Connectors establish the composition by connecting ports. The composition of components results in higher-level components. As an assumption for the sequel of this paper, a higher level component represents just a cluster of its subcomponents and does not add any additional behavior. Thus, all ports that were not connected to other subcomponents have to be delegated to ports of the parent component.

Enhancing ADLs with the notion of variability results in new kinds of modeling elements on different levels of granularity.

• Optionality results in a distinction in mandatory and optional components.
• XOR variability of components introduces a new hierarchical relationship between components: A component variation point represents a generalization of alternative components as it exhibits their commonalities with respect to the architectural role which the alternatives have to fulfill.
• Horizontal connectors, which describe communication and control flows between components, can be distinguished between mandatory and optional connectors. Delegation connectors, which are used to relate two hierarchical levels, cannot exhibit own variability because of our definition of the composition of components.
<table>
<thead>
<tr>
<th>Origin of variability</th>
<th>Occurrence</th>
<th>Signal set</th>
<th>Kind of ports wrt. variability</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent</td>
<td>always</td>
<td>fixed</td>
<td>mandatory, fixed port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>mandatory</td>
<td>varying port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sometimes</td>
<td>fixed</td>
<td>optional, fixed port</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>varying</td>
<td>optional, varying port</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>always</td>
<td>fixed</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>dependently</td>
<td>varying port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sometimes</td>
<td>fixed</td>
<td>dependently optional, fixed port</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>vary</td>
<td>dependently optional, varying port</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Variability of ports.

- Parameterization is applicable on all architectural elements, e.g. parameterized components, ports, connectors, or signals, whereby variability is supported for their attributes.

The consistent integration of the variability concepts yields components with varying interfaces in the form of optional vs. mandatory ports, and in the form of ports with fixed or varying signal sets. Because of the hierarchical composition of components, it can be distinguished whether the variability of a port originates from another, lower level port, i.e. whether the port is independent or dependent: If an optional port of a subcomponent is delegated to its parent component, the corresponding port of the parent is called dependent. The same effect applies to a delegated mandatory port of an optional subcomponent (e.g. the mandatory input port pinSensorsPf of the optional function CbsWdParticleFilter is delegated as a dependently optional port of function Cbs in Figure 2).

The variability of ports can be fully characterized by three variability criteria: occurrence of a port, signal set of a port and origin of variability of a port. The combination of these criteria results in different kinds of ports concerning their variability as summarized in Table 1. Mandatory, fixed ports represent ports already known from common ADLs, where no variability concept is explicitly integrated.

The integration of variability concepts introduces the need to consistently configure all variable elements occurring within an architecture description. The notions dependent or independent as discussed with respect to ports represent an additional specification means to classify occurring variability. The information whether an element is dependent can be automatically computed, and supports the user by determining the minimal set of configuration points to gain a complete configuration. For example, a mandatory, fixed port needs not be configured, because its presence is determined by the presence of the component it belongs to. Thus, there is no dependent version of mandatory, fixed ports in VEIA models.

3 Application example

In the following we illustrate our modeling approach with the help of a case study from our project partner BMW. It describes a distributed support functionality in vehicles called “condition-based service” (CBS) [10]. As the name of the functionality already suggests, maintenance intervals can be computed related to the conditions of the inspected parts. Such a functionality offers the possibility to optimize the schedule of inspection dates and not to solely rely on fixed durations of time or driving distances. Motor oil quality, brakes (brake pads and disks), spark-plugs etc. are examples for relevant wearing parts of a vehicle that were considered within the case study.

The variability used to define the CBS product line is primarily caused by different business service concepts for specific vehicle types, different infrastructures of supported vehicle types,
or by organizational and development process-related aspects. It is reflected in a varying information basis for CBS and in different algorithms\(^1\) to estimate the wearout, for example. With respect to the fact that it is a distributed functionality, i.e. parts of the functionality are allocated to different ECUs in the power train, the development of this functionality as a product line has to cope with the task to find the optimal solution strategy of the management of the variability to be supported.

The definition of a (simplified) CBS product line is shown in Figure 2. The left-hand side of Figure 2 represents a feature model describing the supported variability of the CBS functionality. As an example the variability concerning the computation of the wearout of spark plugs is represented by the feature \(WDSparkPlugs\). This feature, which represents an exclusive alternative, has two children \(WDSparkPlugsAdaptive\) and \(WDSparkPlugsLinear\). Both children represent a functional alternative which cannot be realized jointly in one single product.

Additionally, the left-hand side of Figure 2 depicts a valid configuration of a product where the selection of a feature is indicated by a flag, the de-selection by an x mark. The product currently selected in Figure 2 consists of an Otto engine, where the CBS function monitors the wearout of spark plugs in an adaptive way.

The corresponding function net is shown on the right-hand side of Figure 2.\(^3\) Each wearing part is represented by a corresponding function to compute the wearout. Since the product line supports two ways of computing the wearout of spark plugs (adaptive and linear), the alternative sub-functions \(CbsWdSparkPlugsLinear\)\(^4\) and \(CbsWdSparkPlugsAdaptive\) are introduced. The adaptive variant needs additional input from the spark plugs sensors for the computation. This is represented by an additional, variant-specific port at the \(CbsWdSparkPlugsAdaptive\) function, which becomes a dependent port at the functional variation point \(Cbs\) (port \(pinSensorsSp\)). Furthermore, the wearout detection function for spark

---

\(^1\)linear or adaptive
\(^2\)called P4 in section 5
\(^3\)Some remarks to our graphical notation of function nets: A **mandatory function** is represented as a rectangle. Inner rectangles denote sub-functions. A **function variation point** is represented by a rounded rectangle, its alternatives are the direct sub-functions. Dashed rectangles mark functions as **optional**. The triangle within the port icon represents the communication direction: **in** or **out**. The meaning of the various icon shapes of ports are defined in Table 1.

\(^4\)"wd" is the abbreviation of "wearout detection".
plugs as well as for the particle filter are optional since only vehicles equipped with an Otto engine have spark plugs. Analogously, only vehicles with Diesel engines have particle filters. Both wearout detection functions could have been modeled as alternatives in the function net, but we have decided to model them just as options, because they do not represent “functional” alternatives. In general, the definition of XOR variation points or the usage of options is always a design decision concerning specific objectives and requirements.

We use the feature model as a means to configure the variability occurring within the function net (see the selection marks within the function net in Figure 2). For this purpose we need to link features with the corresponding functions or other architectural elements defining the conditions under which an optional or an alternative architecture element is selected. For instance, we have linked the feature Otto with the function SensorsSparkPlugs stating, that in every vehicle with Otto engine the sensor function for spark plugs has to be present.

The semantical interpretation of the mapping shown in Figure 2 and the formal representation of the product line variance as a feature model are necessary prerequisites to formally analyze the defined product line for consistency, for example. Which properties besides consistency can be analyzed and how such a formal analysis is executed is sketched in the next section.

4 Analysis of feature models

The formal analysis of the before mentioned and introduced development artifacts, feature models and function nets, relies on a representation of the variance of these artifacts in a formal language. There are a lot of proposals in literature to formally analyze variance expressed by feature models. Most of them translate a feature model into a propositional logic formula. Prominent examples are Feature Modeling Plugin (FMP) [20], pure::variants [17] or the FAMA Framework [4]. Using this formal analysis techniques certain properties of feature models that were related to the following questions or operations can be examined:

1. Does the feature model has at least one valid configuration (satisfiability) and if so, how many models (configurations) are represented by that feature model?
2. Does the feature model contain features that were never part of a valid configuration (dead feature detection)?
3. The operation to derive a (partly) configured product out of a feature model is most important during the development process and strongly connected to the binding of variance.
4. If a property such as satisfiability cannot be established, then the user should not simply get the answer “no”, but should get a helpful explanation on the basis of his feature model to be able to analyze the fault and repair it.
5. The ability to prove arbitrary properties about a feature model is concerned with nearly all the before mentioned operations and questions. It gives the user the possibility to formally examine his feature model to ensure the desired feature model behavior on a formal, objectifiable and repeatable basis.

The VEIA project focuses on the one hand on the formal analysis that constitutes a basis to answer all of the above enumerated questions. The VEIA project provides a proof of concept in terms of a prototypical implementation of a feature modeling tool that is able to answer these questions and is not limited to Horn-formulae, for example. The tool realizes a transformation approach that translates a given feature model into a propositional logic or first-order logic formula. This approach allows us to define arbitrary conditions on features that are expressible in the respective logic. These conditions represent constraints on the corresponding feature model that have to be respected if satisfiability is checked or within a configuration process of the respective artifact. The automatic theorem prover SPASS [15] is used as reasoning engine. Using this theorem prover we expect to scale up with huge feature models, since many techniques used to implement for example a constraint propagation mechanism are...
already successfully used as proving strategies within such theorem provers. Furthermore, the integration into a complete development process that is concerned with different refinement levels (see Figure 1) is realized within the VEIA approach by connecting different layers using mappings as illustrated in Figure 2. Such a mapping gives us on the one hand the possibility to trace changes over the complete development process and on the other hand allows for an automatic computation of model configurations.

On the other hand it is important not only to formally analyze variability but also to assess the defined product lines with the help of metrics as described in section 5. Using metrics it is possible to decide which development strategy (product line or 150 percent model) causes the minimal development efforts. Related to the formal analysis of product line specifications the assessment assumes that

- the number of products is computed exactly\(^5\) incorporating all relevant constraints,
- the product line and all relevant products, generated during a configuration process that respects all constraints, are consistent and
- dead features are detected to prevent wrong assessment results.

In our approach we use feature models and the corresponding operations defined on them as the central variability management engine. The integration of this engine into a system development process is one of the major tasks for an enterprise wide consistent and non-isolated variability management approach. To this end we use a translation process [13] that integrates the development artifacts from different layers into one single feature model\(^6\). Within the computation of this model the variability analysis presented in section 2 is used to formulate the respective constraints on functions, ports and their connections.

5 Assessment of product line architectures using metrics

The assessment of a product line architecture relies on a precise description of the system under development. It constitutes the basis for analyses and measurements to assess architectures with respect to certain quality attributes. An important question arising during a product line development is how to implement the variability specified using feature models and function nets in software. This question is tightly related to the point of time within an engineering process at which the variability should be bound: At design time, coding time, compilation time, flash time, or run-time, for instance? To answer this question the product line’s variability has to be investigated, and different design alternatives dealing with this variability need to be compared.

The following two criteria which determine some of the limiting factors in automotive E/E developments are important for the mentioned comparison: the development effort to implement the functionality, and the software code size which determines the memory needs in ECUs as well as the time to flash the code. Metrics which measure the complexity e.g. by function point methods\(^7\) are useful to assess specified product line architectures. For example, the PSSF metrics\(^7\) [12] estimate a relative effort for implementing the specified systems. To this end, the usage frequency of variable components with respect to the considered products needs to be determined. Furthermore, the metrics can also be used to estimate the size of the product’s software code. In general, a prerequisite for a useful application of metrics is the adaptation or calibration of the metrics with respect to the used architecture description languages (matching the specification concepts) as well as to domain and organizational development aspects which influence the development.

Within the CBS case study (see section 3) two alternative software architectures for the implementation the CBS product line are considered: The product line software architecture

---

\(^5\)not only a combinatorial over estimation
\(^6\)used as an internal computation and analysis model
\(^7\)“PSSF” stands for “process-oriented software systems families”
Table 2: Products, frequency of occurrence, and implementation alternatives.

<table>
<thead>
<tr>
<th>Feature</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>Freq. of occ. of Feature in CbsSw150</th>
<th>CbsSwPl</th>
</tr>
</thead>
<tbody>
<tr>
<td>WdMotorOilAdaptive</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>WdParticleFilter-Adaptive</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✔</td>
<td>–</td>
<td></td>
<td>1,025</td>
</tr>
<tr>
<td>WdSparkPlugsAdaptive</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✔</td>
<td>–</td>
<td></td>
<td>1,017</td>
</tr>
<tr>
<td>WdSparkPlugsLinear</td>
<td>–</td>
<td>–</td>
<td>✔</td>
<td>–</td>
<td>–</td>
<td></td>
<td>1,017</td>
</tr>
<tr>
<td>Otto</td>
<td>–</td>
<td>–</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>0,5</td>
</tr>
<tr>
<td>Diesel</td>
<td>✔</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td>0,5</td>
</tr>
</tbody>
</table>

CbsSw150 represents a *product generic approach*[^8] of implementing the product line’s variability. Hereby, standardized components are implemented which fit to any product in the product line, but usually they include functionality not needed in any product. Second, the product line software architecture CbsSwPl represents a *product specific approach*. Hereby, only specific components are built for each product (ideally by generation technologies) which fit “perfectly” to its product. The drawback might relate to logistic and maintenance complexity here. Both alternatives are illustrated in Figure 1, cf. the software architecture artifact. In the following we sum up the results (see Table 3) and activities of applying the metrics on the both software architectures:

1. With the help of the feature model it is defined what products (vehicle types) are built on the basis of the product line (e.g. P1, ..., P5). Then, the products need to be weighted, e.g. how many vehicle types are built with an Otto engine. In our example we assume that 50% of all vehicle types of the product line should be vehicle types with an Otto engine. Because there are three types with Otto engines (P2, P3, P5), we additionally assume that their frequency of occurrence should be the same (16,7%), cf. Table 2. Based on this information, the frequency of occurrence of single features can be computed for a product line: For example, the value for WdMotorOilAdaptive is 100% because the feature is present in all products; the feature WdParticleFilterAdaptive gets the value 25% because it's only present in product P5.

2. Now, the alternative software solutions (CbsSw150 vs. CbsSwPl) can be analyzed: The frequency of occurrence for the software components can be computed on the basis of the corresponding features. In case of CbsSwPl, the computation of the usage frequency of components is straight-forward: A software component which realizes the WdParticleFilterAdaptive feature is installed and used in 25% of all products. In case of the product generic approach CbsSw150, a corresponding software component is installed in all products (100%). The result of this analysis is shown in Table 2.

3. To estimate the development efforts and costs, the metrics have to be applied on the architecture models of the alternative software solutions, based on the above assumptions. In order to estimate the software code size, the single product architectures of the different solution strategies need to be analyzed. The results for the example are shown in Table 3:

---

[^8]: Such an approach is often called “150 percent approach”.
[^9]: Software components related to this actually deselected feature are still implemented and installed within the product of the CbsSw150 product line (unless the ECU for that software component is not present in the vehicle). But such software components may be deactivated during run-time (e.g. dead code). Because of that the frequency of occurrence for the feature is “1” (100%) in CbsSw150, i.e. it is “present” in every product of the CbsSw150 product line.
Table 3: Metrics applied on software architecture specifications.

<table>
<thead>
<tr>
<th></th>
<th>Generic (CbsSw150)</th>
<th>Specific (CbsSwPl)</th>
<th>Products of CbsSwPl</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. components</td>
<td>14</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>No. ports</td>
<td>38</td>
<td>38</td>
<td>19</td>
</tr>
<tr>
<td>development effort</td>
<td>178</td>
<td>236</td>
<td>19</td>
</tr>
<tr>
<td>software code size of a product</td>
<td>168</td>
<td>122</td>
<td>110</td>
</tr>
</tbody>
</table>

The metrics state that the product generic approach is better than the specific one regarding the effort: 178 vs. 236 (or 222) points. But, regarding code size (e.g., because of flash time constraints) the product specific approach seems to be better: The products on the basis of the specific approach have an indicated code size of averaged 122 points vs. 168 points in case of the generic implementation approach.

As proof of concept, the metrics were also applied on more elaborated specifications of the case study “Condition-Based Service”, cf. [10, 9], where the introduced approach is discussed in more detail. The mentioned metrics are prototypically implemented within v.control as illustrated in Figure 3.

6 Conclusion and related work

The presented method introduces the possibility to connect different layers within a structured development process in a generic way resulting in an effective approach to configure and to compute products according to predefined measures. It supports the user in finding valid configurations and guarantees that the constraints are not violated. The proposal is flexible in the sense that it allows to incorporate variance that is introduced at later stages of the development without changing the before specified development artifacts. Those artifacts that are specified and defined below the level of the functional architecture can be integrated analogously resulting in a pervasive development of variability throughout the complete product development process.

As proof of concept the work is prototypically implemented in a case tool called v.control. It demonstrates different aspects of our methodology as there are the specification of product lines.
in form of function and software component architectures including abstraction and reuse issues, the assessment of a product line by the evaluation of the architecture specification using metrics (see Figure 3), and the consistent configuration of a product line architecture.

ADLs are essential means for a system and software design in the large. An overview and a taxonomy for comparison of ADLs is given in [14]. ADLs for product lines often introduce structural variability for components, but only deal with constant, maybe optional, interfaces for components. A comparison of those ADLs is given in [1, 18]. [6, 7, 2] are examples of ADLs for automotive or embedded systems engineering. Several aspects of product lines are discussed in e.g. [16, 19, 3]. Exploiting feature models for variability management is done by e.g. [20, 17]. pure::variants [17] uses a generic family model as abstraction of blocks of a target language description (e.g. source code). By configuration of a feature model, the family model is used to concatenate the blocks to build a product. Thus, the presented architectural models can be regarded as specializations of the family model.

References