Control Mechanisms for Managed Evolution of Automotive Software Product Line Architectures

Christoph Knieke, Marco Körner, Andreas Rausch,
Mirco Schindler, Arthur Strasser, and Martin Vogel
TU Clausthal, Department of Computer Science, Software Systems Engineering
Clausthal-Zellerfeld, Germany
Email: {christoph.knieke|marco.koerner|andreas.rausch|
mirco.schindler|arthur.strasser|m.vogel}@tu-clausthal.de

***Abstract*—The high time and cost pressure in the automotive market encourages reuse of components and software in different vehicle projects leading to a high degree of variability within the software. Often, a product line approach is used to handle variability. However, the increasing complexity and degree of variability of automotive software systems hinders the capabilities for reusability and extensibility of these systems to an increasing degree. After several product generations, software erosion is growing steadily, resulting in an increasing effort of reusing software components, and planning of further development. Here, we propose control mechanisms for a managed evolution of automotive software product line architectures. We introduce a description language and its meta model for the specification of the software product line architecture and the software architecture of the corresponding products. Based on the description language we propose an approach for architecture conformance checking to identify architecture violations as a means to prevent architecture erosion. We demonstrate our methodology on a real world case study, a brake servo unit (BSU) software system from automotive software engineering. To show the benefits of our approach, we define several metrics on architecture and software level and apply the metrics on the BSU example.**

Keywords–Architecture Conformance Checking; Architecture Description Language; Software Product Lines; Automotive Software Engineering.

I. INTRODUCTION

This paper is a substantial extension of the work presented at the ADAPTIVE 2017 conference [1]. In the development of electronic control unit (ECU) software for vehicles, the reduction of development costs and the increase of quality are essential objectives. A significant measure to achieve these goals is the reuse of software components [2]. The reuse is mainly achieved by a product-wide development for different vehicle variants: Different configurations of driver assistance systems, comfort functions, or powertrains can be variably combined, creating an individual and unique product. Furthermore, for each new vehicle generation, the software of preceding generations of the vehicle is reused or adopted [3].

However, the possibilities for reuse and extensibility of existing functions can not be fully exploited in many cases. Rather, it can be observed that due to the increase in so-called "accidental" complexity [3] (see Section VI-B), the reusability and further developability reaches its limits. One reason for this is the lack of a product-line-oriented overall planning, based on the concepts of software product line engineering already established in other domains. A central factor here is the planning based on a product line architecture (PLA), on the specification of which the individual products are derived. The PLA describes the structure of all realizable products. Each product that is developed has an individual product architecture (PA) whose structure should be mapped onto the PLA.

An important challenge with regard to the architecture is to minimize architecture erosion as illustrated in the following. In [4], architecture erosion is defined as "*the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture.*" As shown in Figure 1 a PLA is designed initially and develops over time [5]. It makes no difference whether the PLA is explicitly planned or exists only implicitly in the minds of the participants. In the further development, it must be ensured that the product architecture remains compliant with the product line architecture. However, due to the high time and cost pressure in the automotive sector, it is not possible for every further development to be controlled via the product line. Rather, some product-specific adjustments have to be made. This can lead (intentionally or unintentionally) to a product architecture that differs in comparison to the product line architecture: the architecture erodes. In the long-term, this leads to reduced reusability and extensibility of the software artifacts.

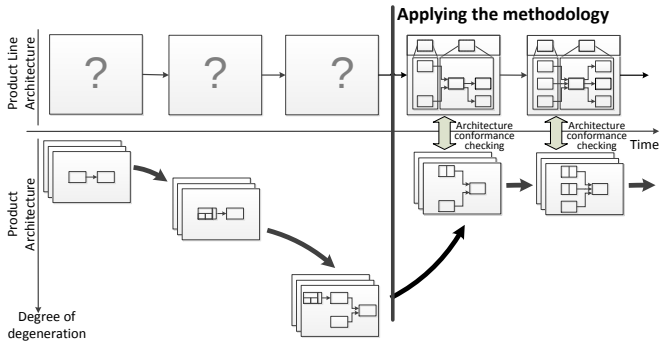

Figure 1. Avoiding architecture erosion by applying our methodology

To prevent architecture erosion, we propose control mechanisms for a managed evolution of automotive software product line architectures in this paper. We refer to the results shown in a preceding paper [5] to create a PLA as a prerequisite for our approach by applying strategies for architecture recovery and discovery. First, we introduce a description language and its meta model for the specification of the software product line architecture and the software architecture of the corresponding products. Based on the description language we propose an approach for architecture conformance checking to identify architecture violations as a means to prevent architecture

erosion (indicated on the right side of Figure 1). Due to the size of the product line architecture, an automated consistency check is necessary, which is an essential part of our approach to counteract architecture erosion.

The application of the software product line development must take into account the special properties, boundary conditions and requirements that exist in the automotive environment [6]. Therefore, a method adapted to the automotive environment is required and is presented in this paper.

An important aspect is the design and planning of further developments of the product line architecture. When designing the product line architecture, the architecture must be based on architecture principles appropriate for the automotive domain, aiming at reusability and further development [3]. Since a wide range of products can be affected by the further development of the product line architecture, changes must be carefully planned: High demands are placed on the reliability of the systems, but the reliability is endangered by extensive adaptations.

The major objectives of our approach can be summarized as follows:

1) Maintaining stability of the PLA and minimizing product architecture erosion even if extensive further development of the system takes place.
2) Achieving high scalability, and a high degree of usage of the modules.

The first objective primarily addresses the software architecture: The PLA should be based on appropriate design principles that allow further developments with a minimal adjustment effort of the PLA. At the same time, the erosion of product architecture is to be minimized.

The second objective focuses on the software components: These should be kept scalable so that they can be used for as many variants as possible. Nevertheless, the software components should be able to be reused over time in subsequent product generations. However, the high variability within the components increases the complexity of the components and thus makes reuse more difficult.

The paper is structured as follows: Section II gives an overview on the related work. In Section III we propose a methodology for managed evolution of automotive software product line architectures. Section IV introduces an architecture description language for the specification of the product line architecture and the product architecture. Based on this description language Section V proposes an approach for architecture conformance checking. In Section VI, we apply our approach on a real world example, a brake servo unit, from automotive software engineering. The results of a corresponding field study are evaluated and discussed in Section VII. Section VIII concludes.

II. Related Work

To the best of our knowledge, no continuous overall development cycle for automotive software product line architectures exists. Several aspects of our process are already covered in literature:

A. Reference Architectures

The purpose of the reference architecture is to provide guidance for future developments. In addition, the reference architecture incorporates the vision and strategy for the future. The work in [7] examines current reference architectures and the driving forces behind development of them to come to a collective conclusion on what a reference architecture should truly be. Furthermore, in [7], reference architectures are assumed to be the basis for the instantiation of product line architectures (so-called family architectures, see [7]).

Nakagawa et. al. discuss the differences between reference architectures and product line architectures by highlighting basic questions like definitions, benefits, and motivation for using each one, when and how they should be used, built, and evolved, as well as stakeholders involved and benefited by each one [8]. Furthermore, they define a reference model of reference architectures [9], and propose a methodology to design product line architectures based on reference architectures [10][11].

B. Software Erosion

In [4], de Silva and Balasubramaniam provide a survey of technologies and techniques either to prevent architecture erosion or to detect and restore architectures that have been eroded. However, each approach discussed in [4] refers to architecture erosion for a single PA, whereas architecture erosion in software product lines are out of the scope of the paper. Furthermore, as discussed in [4], none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

Van Gurp and Bosch [12] illustrate how design erosion works by presenting the evolution of the design of a small software system. The paper concludes that even an optimal design strategy for the design phase does not lead to an optimal design. The reason for this are unforeseen requirement changes in later evolution cycles. These changes may cause design decisions taken earlier to be less optimal.

The work in [13] describes an approach to flexible architecture erosion detection for model-driven development approaches. Consistency constraints expressed by architectural aspects called architectural rules are specified as formulas on a common ontology, and models are mapped to instances of that ontology. A knowledge representation and reasoning system is then utilized to check whether these architectural rules are satisfied for a given set of models. Three case studies are presented demonstrating that architecture erosion can be minimized effectively by the approach.

C. Software Product Line Architectures

As discussed in [5] an overall automotive product line architecture is often missing due to software sharing. Thus, architecture recovery and discovery has to be applied by concepts of software product line extraction [5]. The aim of software product line extraction is to identify all the valid points of variation and the associated functional requirements of component diagrams. The work in [14] shows an approach to extract a product line from a user documentation. The Product Line UML-based Software Engineering (PLUS) approach permits variability analysis based on use case scenarios and the specification of variable properties in a feature model [15]. In [16], variability of a system characteristic is described in a feature model as variable features that can be mapped to use cases. In contrast to our approach, these approaches are based

on functional requirements whereas our approach is focused on products.

In numerous publications, Bosch et. al. address the field of product line architecture, software architecture erosion, and reuse of software artifacts: The work in [17] proposes a method that brings together two aspects of software architecture: the design of software architecture and software product lines. Deelstra et al. [18] provide a framework of terminology and concepts regarding product derivation. They have identified that companies employ widely different approaches for software product line based development and that these approaches evolve over time. The work in [19] discusses six maturity levels that they have identified for software product line approaches. In [20], a methodical and structured approach of architecture restoration in the specific case of the brake servo unit (BSU) is applied. Software product lines from existing BSU variants are extracted by explicit projection of the architecture variability and decomposition of the original architecture.

The work in [21] gives a systematic survey and analysis of existing approaches supporting multi product lines and a general discussion of capabilities supporting multi product lines in various domains and organizations. They define a multi product line (MPL) as a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system. The different product lines in an MPL can exist independently but typically use shared resources to meet the overall system requirements. According to this definition, a vehicle system is also an MPL assuming that each product line is responsible for a particular subsystem. However, in the following, we only regard classic product lines, since the dependencies between the individual product lines in vehicle systems are very low, unlike MPL.

D. Software Product Line Architecture Evolution

Thiel and Hein [22] propose product lines as an approach to automotive system development because product lines facilitate the reuse of core assets. The approach of Thiel and Hein enables the modeling of product line variability and describes how to manage variability throughout core asset development. Furthermore, they sketch the interaction between the feature and architecture models to utilize variability.

Holdschick [23] addresses the challenges in the evolution of model-based software product lines in the automotive domain. The author argues that a variant model created initially quickly becomes obsolete because of the permanent evolution of software functionalities in the automotive area. Thus, Holdschick proposes a concept how to handle evolution in variant-rich model-based software systems. The approach provides an overview of which changes relevant to variability could occur in the functional model and where the challenges are when reproducing them in the variant model.

Automotive manufacturers have to cope with the erosion of their ECU software. The work in [3] proposes a systematic approach for managed and continuous evolution of dependable automotive software systems. It is described how complexity of automotive software systems can be managed by creating modular and stable architectures based on well-defined requirements. Both architecture and requirements have to be managed in relation. Furthermore, to face the lack of flexibility of existing hieratic automotive software systems development approaches, they are focusing on four driving factors: systems engineering and agile function development, feature and function driven team development, agile management principles, and a seamless tooling infrastructure supporting continuously and iteratively evolving automotive software systems in a flexible manner.

To counteract erosion it is necessary to keep software components modular. But modularity is also a necessary attribute for reuse. Several approaches deal with the topic reuse of software components in the development of automotive products [2][24]. In [2], a framework is proposed, which focuses on modularization and management of a function repository. Another practical experience describes the introduction of a product line for a gasoline system from scratch [24]. However, in both approaches a long-term minimization of erosion is not considered.

A previous version of our approach is described in [5] focusing on the key ideas of the management cycle for product line architecture evolution. Furthermore, an approach for repairing an eroded software consisting of a set of product architectures by applying strategies for recovery and discovery of the product line architecture is proposed.

III. Overall Development Cycle

Our methodology for managed evolution of automotive software product line architectures is depicted in Figure 2. The left part of Figure 2 depicts the recovery and discovery activity introduced in a previous paper [5]. This activity is performed once before the long term evolution cycle (right side of Figure 2) can start. The latter consists of two levels of development: The cycle on the top of Figure 2 constitutes the development activities for product line development, whereas the second cycle is required for product specific development. Not only both levels of development are executed in parallel but even the activities within a cycle may be performed concurrently. The circular arrow within the two cycles indicates the dependencies of an activity regarding the artifacts of the previous activity. Nevertheless, individual activities may be performed in parallel, e.g., the planned implementations can be realized from activity `PL-Plan`, while a new PLA is developed in parallel (activity `PL-Design`). The large arrows between the two development levels indicate transitions requiring an external decision-making process, e.g., the decision to start a new product development or prototyping, respectively.

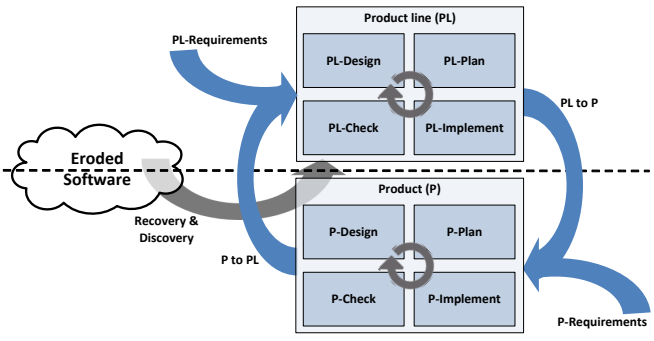

Figure 2. Overall development approach

In the following three subsections, we will explain the basic activities of our approach in detail by referring to the terms depicted in Figure 2. Table I gives a brief overview on the

TABLE I. Explanation of the activities in Figure 2.

Activity	*Input*	*Objective*	*Output*
PL-Design	Software system / component requirements and documentation from product development.	Further development of PLA with consideration of design principles. Application of measuring techniques to assess quality of PLA.	New PLA (called "PLA vision").
PL-Plan	PLA vision.	Planning of a set of iterations of further development toward the PLA vision taking all affected projects into account.	Development plan including the planned order of module implementations and the planned related projects.
PL-Implement	Development plan for product line.	Implementation including testing as specified by the development plan for product line development.	Implemented module versions.
PL-Check	Architecture rules and set of implemented modules to be checked.	Minimization of product architecture erosion by architecture conformance checking based on architecture rules.	Check results.
P-Design	Project plan and product specific requirements.	Designing product architecture and performing architecture adaptations taking product specific requirements into account. Compliance checking with PLA to minimize erosion.	Planned product architecture.
P-Plan	Product architecture.	Definition of iterations to be performed on product level toward the planned product architecture.	Development plan for product development.
P-Implement	Development plan for product development.	Product specific implementations including testing as specified by the development plan for product development.	Implemented module versions.
P-Check	Architecture rules and set of implemented modules to be checked.	Architecture conformance checking between PLA and PA.	Check results.
PL to P	Development plan for product line.	Defining a project plan by selecting a project from the the product line.	Project plan.
P to PL	Developed product.	Providing product related information of developed product for integration into product line development.	Product documentation and implementation artifacts of developed products.
PL-Requirements	Requirements.	Specification and validation of software system and software component requirements by requirements engineering.	Software system and software component requirements.
P-Requirements	Requirements in particular from calibration engineers.	Specification of special requirements for a certain vehicle product including vehicle related parameter settings.	Vehicle related requirements.
Recovery & Discovery	Source artifacts (developed products).	Recovery of the implemented PLA from the source artifacts (developed products) and discovery of the intended PLA.	Implemented and intended PLA.

objectives of each of the 13 activities, including inputs and outputs.

We distinguish between the terms 'project' and 'product' in the following: A project includes a set of versioned software components, so-called modules. These modules contain variability so that a project can be used for different vehicles. On the other hand, a product is a fully executable software status for a certain vehicle based on a project in conjunction with vehicle related parameter settings.

A. Planning and Evolving Automotive Software Product Line Architectures

(`PL-Requirements`) Software system requirements and software component requirements from requirements engineering serve as input to the management cycle of the PLA. Errors occurring during the phase of requirements elicitation and specification have turned out to be major reasons for the failure of IT projects [25]. In particular, errors occur in case the requirements are specified erroneous or the requirements have inconsistencies and incompleteness. Errors during the phase of requirements elicitation and specification can be avoided by choosing an appropriate specification language enabling the validation of the requirements. In [26], e.g., activity diagrams are considered for the validation of system requirements by directly executable models including an approach for symbolic execution and thus enabling validation of several products simultaneously.

(`P to PL`) Artifacts of the developed product from the product cycle in Figure 2 serve as further input to the management cycle of the PLA: The product documentation contains architectural adaptations and change proposals, which can be integrated in the PLA. Furthermore, the modified modules in their new implementation are committed to the management cycle of the PLA for integration in product line.

(`PL-Design`) Next, we consider the design of the PLA. Generally, a software system architecture defines the basic organization of a system by structuring different architectural elements and relationships between them. The specification of "good" software system architecture is crucial for the success of the system to be developed. By our definition, a "good" architecture is a modular architecture that is built according to the following: (a) design principles for high cohesion, (b) design principles for abstraction and information hiding, and (c) design principles for loose coupling. In [3], we propose methods and techniques for a good architecture design. Based on these methods and techniques a new PLA is defined (called PLA vision) taking the new requirements (`PL-Requirements`) and product related information (`P to PL`) into account. To assess the quality of the designed PLA, it is necessary to measure complexity and to describe the results numerically. In particular, we consider properties such as cohesion, coupling, reusability and variability in order to draw conclusions about the quality of the PLA.

(`PL-Plan`) As further development of the PLA will effect a high number of products, the changes have to be planned carefully in order to avoid errors within the corresponding products and to avoid architecture erosion. Thus, the planning phase has to define a set of iterations of further development towards the PLA vision. All allowed changes are planned as a schedule containing the type of change and timestamp. It is planned, in which order the implementation of corresponding modules should take place. It should be emphasized that there are many parallel product developments, which must be taken into account when planning. Next, either affected projects and modules are determined or a pilot project is selected.

Some further developments can lead to extensive architectural changes. In this case the effects of the architectural changes on the associated projects have to be closely examined. For this purpose further development projects can be defined as prototype projects for certain iterations of the PLA. These projects are then tested within the product cycle.

B. Automotive Product Development and Prototyping

(PL-Implement) The former planning activity has determined the schedule for PLA adaptations and product releases. Thus, on the implementation level, new versions of the software are planned, too. Vehicle functions are modeled using a set of modules, specifying the discrete and continuous behavior of the corresponding function. As required by ISO 26262 [27], each module needs to be tested separately. Established techniques for model-based testing necessitate a requirements specification, from which a test model can be derived. In practice, requirements are specified by natural language and on the level of whole vehicle functions instead of modules so that test models on module level can not be derived directly. Therefore, in [28], a systematic model-based, test-driven approach is proposed to design a specification on the level of modules, which is directly testable. The idea of test-driven development is to write a test case first for any new code that is written [29]. Then the implementation is improved to pass the test case. Based on the approach in [28] we use the tool Time Partition Testing (TPT) because it suits particularly well due to the ability to describe continuous behavior [30]. The modules may be developed in ASCET or MATLAB/Simulink.

(P-Requirements) Releasing a fully executable software status for a certain vehicle product requires a specification of vehicle related parameter settings. Furthermore, special requirements for a specific product may exist necessitating further development of certain implementation artifacts. Building an executable software status for a certain vehicle product is realized by the cycle at the bottom of Figure 2. In contrast, the product line cycle in Figure 2 includes the development of sets of software artifacts of all planned projects.

(PL to P) Automotive software product development and prototyping starts with selecting a product from the product line. Therefore, the project plan is transferred containing module descriptions and descriptions of the logical product architecture integration plan with associated module versions.

(P-Plan) The product planning defines the iterations to be performed. An iteration consists of selected product architecture elements and planned implementations. An iteration is part of a sequence of iterations.

(P-Implement) An iteration is completed when all planned elements of an iteration are implemented according to the test-driven approach of [28].

C. Architecture Conformance Checking

Architecture erodes when the implemented architecture of a software system diverges from its intended architecture. Software architecture erosion can reduce the quality of software systems significantly. Thus, detecting software architecture erosion is an important task during the development and maintenance of automotive software systems. Even in our model-driven approach where implementation artifacts are constructed w.r.t. a given architecture the intended architecture and its realization may diverge. Hence, monitoring architecture conformance is crucial to limit architecture erosion.

Each planned product refers to a set of implementation artifacts, called modules. These modules constitute the product architecture. The aim of PL-Check and P-Check is the minimization of product architecture erosion. In [13], a method is described to keep the erosion of the software to a minimum: Consistency constraints expressed by architectural aspects called architectural rules are specified as formulas on a common ontology, and models are mapped to instances of that ontology. Based on this approach we are extracting rules from a PLA to minimize the erosion of the product architecture. During the development of implementation artifacts the rules can be accessed via a query mechanism and be used to check the consistency of the product architecture. Those rules can, e.g., contain structural information about the software like allowed communications. In [13], the rules are expressed as logical formulas, which can be evaluated automatically to the compliance to the PLA.

(PL-Check) After each iteration planned in activity PL-Plan all related product architectures have to be checked. As P-Check refers to one product only, the check is performed after all related implementation artifacts of the product are developed.

(P-Design) The creation of a new product starts with a basically planned product architecture commonly derived from the product line. For the development of the product, new functionalities have to be realized and thus necessary adaptations to the planned product architecture are made. In order to keep the erosion to a minimum we have to ensure the compliance to the architecture design principles of the PLA. Therefore, we check consistency of the planned product architecture by applying architecture rules from the PLA.

However, in the case of prototyping it may be desired that the planned product architecture differs from PLA specifications. Thus, as a consequence, the architecture rules are violated. As pointed out in Section III-A, product related information is returned to the management cycle of the PLA after product delivery. If the development of a product required a differing product architecture w.r.t. the PLA, this could advance the erosion. Necessary changes must be communicated to PL-Design and PL-Plan s.t. the changes can be evaluated and adopted. As changes to the PLA can have severe influences on all the other architectures the changes are not applied immediately but considered for further development.

IV. Architecture Description Language

The software architecture serves as input for the subsequent development steps, e.g., for implementation and test. In this architecture, the software building blocks of the cars embedded system are documented. Thereby, the implementation step follows the model-based development approach, where code is generated from architecture models using tools of the industrial partner. To model these architectures in the industrial projects we introduced the EMAB (Einheitliche Modulare Architektur Beschreibung) architecture description language. The EMAB is applied for the architecture extraction and the managed evolution approach presented in this paper. In addition, the EMAB includes all aspects to describe the static structure of the project partners electronic control unit system domains.

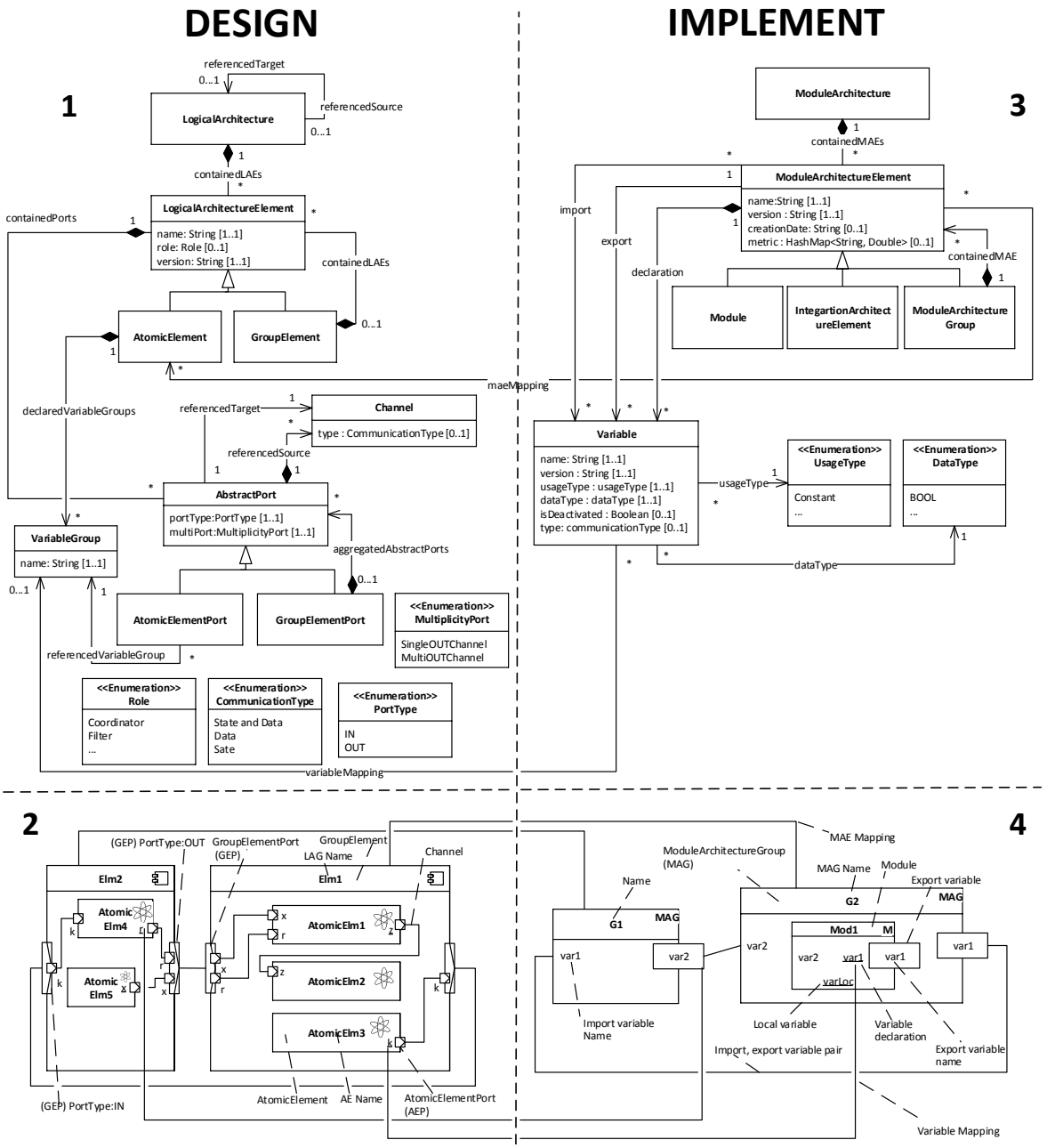

Figure 3. EMAB: The meta model (1,3) and the two views as instances of the meta model (2,4).

EMAB is used to describe two layered architectures consisting of the logical architecture layer called DESIGN and the technical software architecture layer called IMPLEMENT. Both are defined by the syntax and semantics of the EMAB meta model elements. For each layer, the EMAB also defines the appropriate block diagram based views for architecture description. In our approach, the two layers DESIGN and IMPLEMENT refer to activities `PL-Design`/`P-Design` and `PL-Implement`/`P-Implement`, respectively.

The following subsections IV-A, IV-B illustrate the details of the EMAB syntax and appropriate semantics of the DESIGN layer and IMPLEMENT layer and their appropriate views. Figure 3 shows the meta model to illustrate the description language syntax (top) and also shows an exemplary instance of the meta model to illustrate the syntax of the views (bottom). Each layer and view are identified by one of the four quadrants in Figure 3.

A. Meta Model and View of the DESIGN Layer

Logical Architecture Element: The first quadrant of Figure 3 shows the syntax of the logical architecture comprising the building blocks. The `LogicalArchitecture` contains a set of `LogicalArchitectureElements` (LAEs). The LAE is defined by at least the `name` attribute, the `version` attribute and optionally by the `role` attribute. The meta model contains the roles `Coordinator`, `Support`, `Filter`, which are used for several architecture concepts.

There are two types of LAEs: Decomposable GroupElement and not decomposable AtomicElement architecture building blocks. The GroupElement is used to describe a hierarchy of building blocks. A GroupElement can be decomposed into smaller LAEs.

The second quadrant of Figure 3 shows the DESIGN view by a meta model instance example of the logical architecture. The diagram represents a logical architecture consisting of two GroupElement building blocks and five AtomicElement building blocks. The block named Elm1 represents a GroupElement and the building block AtomicElm1 represents an AtomicElement. The hierarchy is shown by nesting of the AtomicElement block AtomicElm1 in the GroupElement block Elm1.

Interface of a LAE: The interface of a building block is defined by ports. There are two kinds of ports: Decomposable ports called GroupElementPort (GEP) and not decomposable ports called AtomicElementPort (AEP). GEPs can be used only by GroupElement building blocks whereas AEP ports can be used by both, GroupElement and AtomicElement, building blocks. A GEP of the outer GroupElement block enables the composition of ports of nested inner LAEs. Thereby, several ports of a GroupElement building block can be grouped to one port. The type of information that is used to pass by a port to extern is defined by a VariableGroup. A VariableGroup represents the functional information that abstracts existing interfaces of the IMPLEMENT layer.

The syntax to describe ports of a building block is depicted in the first quadrant of Figure 3. Therefore, the LAE has a containment association aggregatedAbstractPorts to any number of AbstractPorts. The AbstractPort element has to define at least the attribute portType, which is an enumeration type of IN, OUT. GEP and AEP have an inherit association to AbstractPort. Though, an AbstractPort can be associated to zero or one GEP. The AEP has a containment association referencedVariableGroup to any number of VariableGroups. A VariableGroup must be associated to exactly one AtomicElement and can be associated by any number of AEPs.

The second quadrant of Figure 3 shows the block diagram notation of instances of AEP and GEP meta model elements. The right border of the GroupElement block Elm2 has a rectangular block that consists of several smaller squares-shaped blocks. The rectangular block represents a GEP element and the smaller square-shaped blocks represent AEP elements. The GEP composes the two out ports of the AtomicElements AtomicElm4 and AtomicElm5. Thereby, to represent that a AEP owned by the inner AtomicElement is composed by the GEP of the outer GroupElement, the following must be modeled: For each AEP of an inner AtomicElement that has to be composed, a second AEP is described that is contained by the GEP. Both, the AEP contained by the inner AtomicElement and the AEP contained by the GEP must reference the same VariableGroup and must be connected to each other. For example, the two connected OUT AbstractPorts that reference the VariableGroup x in the Elm2 block represent the composed OUT AEP of the AtomicElement AtomicElm5. This modeling step is used to enable a more detailed visualization of the composition of different kinds of information. Connections are modeled by the Channel meta model element, which is introduced in the following.

Channel: The communication in the development of control systems is modeled by data flow. The data flow represents functional information of the communication. Each data flow has a source, a target and forms a point to point connection for a source, target pair. The point to point connection is called Channel. In the industrial projects several kinds of functional information (e.g., states, functional data) have been identified. These represent the types of the Channel.

The Channel meta model element is shown in the first quadrant of Figure 3. This element has an optional attribute type to define the functional information, which can be one of the three CommunicationType items. The items of this enumeration type are State and Data, Data, and State. The Channel has at least one association referencedSource to a source AbstractPort and at least one association referencedTarget to a target AbstractPort.

The connections, which are instances of the Channel meta model element, are depicted in the second quadrant of Figure 3. For example, the connections of ports, which have associations to VariableGroups x,r of AtomicElm1, AtomicElm4, and AtomicElm5, specify the communication of these AtomicElements. These connections comprise the ports of the GroupElements Elm1 and Elm2, as each AtomicElement is contained in a different GroupElement. Therefore, ports of the inner AtomicElements are connected to ports of the outer GroupElements to model the communication path along with the connections of the GroupElements.

B. Meta Model and View of the IMPLEMENT Layer

Module Architecture Element: The building block of the software system is described by the ModuleArchitectureElements (MAE). These building blocks are used to form a module architecture, which describes the structure of the software system. Each MAE is managed in a versioning system and is identified by its name and its version. Moreover, each MAE has a time stamp to identify the last time of modification and a quality degree regarding modularity measuring. The following kinds of MAEs are distinguished:

- Not decomposable building blocks called Module representing code artifacts.
- Building blocks that can be composed are called ModuleArchitectureGroup (MAG) to describe hierarchical structures.
- The IntegratedArchitectureElement that is used to describe building blocks of software subsystems developed by software suppliers.

The third quadrant of Figure 3 shows the syntax of the meta model for the description of the IMPLEMENT layer of the module architecture. The ModuleArchitecture is the root of the architecture that has a containment association containedMAEs to any number of MAEs. The MAE has at least the attributes name, version and has the optional attributes creationDate, metric. The MAG, the IntegratedArchitectureElement and the Module have an inheritance association to MAE. The MAE has the

association containedMAE to any number of MAEs. A MAE can be associated via containedMAE by exactly one MAG.

The fourth quadrant of Figure 3 shows the block diagram syntax as meta model instance example of the view of the IMPLEMENT layer. The diagram represents an instance of the meta model module architecture that contains two MAG instances named G1, G2. Moreover, in the block G2, another block named M1 is nested. M1 represents an instance of the meta model Module element.

Interface - Variable: The interface of a MAE is defined by the set of all import and export variables that can be compared to the imports/exports of an ANSI C header file. Only Module declares its signals called Variables that are used for communication to other MAEs. A Variable is globally known and has a system global unique identifier that is identified by its name and its version and has at least a DataType (e.g., byte) and at least the UsageType. The UsageType is used to describe whether the data of a Variable is constant at run time or not. Moreover, some Variables are used in some software systems but in others not. Therefore, the flag isDeactivated is used to define the presence of a variable declaration. The flag is retrieved at compile time to remove variable declarations from code. Several kinds of variable types for communication have been identified from the functional point of view. For this purpose, a variable can have one of the following CommunicationTypes: State and Data, Data, and State. Another special feature of the MAG is that its interface can only be defined by its inner MAE interfaces. For example, an export interface v1 of a MAG can only be defined when the MAG has an inner MAE that declares variable v1 and exports this variable.

The syntax of the meta model to describe the MAE interface is depicted in the third quadrant of Figure 3. The MAE has several associations to describe the interface. First, the MAE has an association import to a number of Variables. Each import can be associated by a Variable to a number of MAEs. Second, the MAE has an association export to a number of Variables. Each export Variable can be associated by exactly one MAE. Third, the MAE has an association declaration to any number of Variables. Each declaration can be associated by exactly one MAE. For each declared Variable, at least attributes name, version, dataType, usageType have to be defined. The other attributes isDeactiveable und type are optional.

In the fourth quadrant of Figure 3 each diagram element represents an instance of the appropriate meta model element. In the case of the interfaces, the diagram shows three declared variables varLoc, var1, var2. Import variables are shown on the left border side within each block. For example, the MAG G2 has the import variable var2. Export variables are shown on the right border side of each block. Module Mod1 has the export variable var1. Moreover, var1 is declared by Mod1, which is shown by the underlined name of the variable. If a variable is only declared but not used as an interface, then it is used only local by the module. For example, varLoc is used by Mod1 locally.

Communication: The communication in software control systems is modeled as data flow in structures that represent directed graphs. For example, executable models in Mathlab Simulink or ASCET, represent this kind of data flow based graphs. The ModuleArchitecture is used to describe the communication implicitly by a pair of MAE interfaces. An interface pair has to describe a variable that is shared by the pair of an import interface and an export interface of two MAE. These two MAEs describe the same variable for communication from one MAE to another MAE. One MAE must declare the variable as an export interface. The other MAE must have the same variable as an import interface.

The syntax of the meta model to describe the communication is shown in the third quadrant of Figure 3. The communication is described implicitly by two MAE. One MAE must have an association import to a Variable. The other MAE must have an association export to the same Variable as the other MAE.

The fourth quadrant of Figure 3 shows an example model of the view where the communication of G2 and G1 is described by a connection of the export var1 block of G2 to import var1 of G1. Thereby, a connection is shown only in the case where the pair of import interface and export interface belong two inner MAEs of common outer MAE as root architecture element.

Mapping of the implement layer and the design layer: The structure of the logical architecture description of the DESIGN layer must be fulfilled by the module architecture of the IMPLEMENT layer. The EMAB description is called consistent, if the DESIGN layer is fulfilled by the IMPLEMENT layer. Therefore, each AtomicElement and each VariableGroup must have equivalent Variable elements and MAE elements in the IMPLEMENT layers.

The first quadrant and third quadrant of Figure 3 show the syntax of the meta model for the Variable mapping and the MAE mapping. The mapping is necessary to describe equivalent elements of the two layers. The MAE element from the first quadrant has the association maeMapping to any number of AtomicElements. The Variable has the association variableMapping to zero or one VariableGroup. Each VariableGroup is associated by any number of Variables and each AtomicElement to any number of MAEs.

Figure 3 shows the syntax of the view where an instance of a mapping is represented by a connection. The export interface var1 of block G2 and the VariableGroup k are mapped by a connection, which represents an instance of the variableMapping. The connection between block G2 and block Elm1 represents an instance of the maeMapping.

C. Valid Descriptions

From the technical point of view, the EMAB models are stored as XML files. During the export or import the file validity against the XML schema is checked. But not every model that is valid against the XML schema, is also a valid architecture description. Therefore, in the following several rules are introduced for each layer in detail. These rules must be fulfilled to ensure the validity of the description of the two architecture layers. In the following all necessary rules ensuring that EMAB models from the technical point of view are valid XML files are specified in prose. Each rule can be implemented using some constraint language. Figure 7 (see Section V) shows an example of some conformance checking

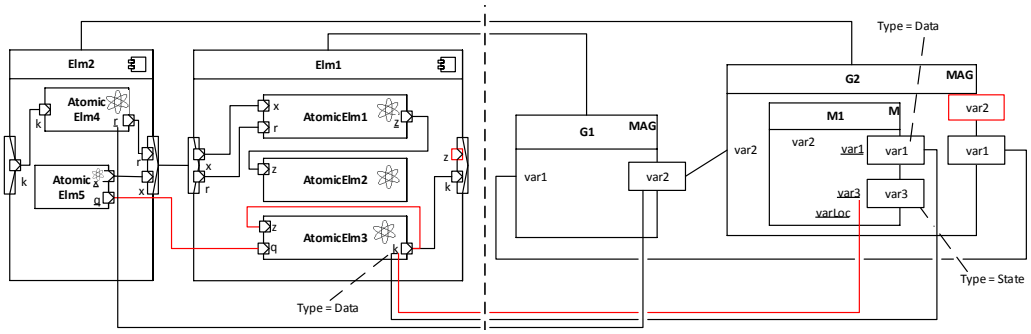

Figure 4. Example of invalid EMAB descriptions

rule that is implemented in the object constraint language (OCL).

Design layer rules: The EMAB description of the logical architecture is valid, if the following rules are fulfilled:

1) A `Channel` can connect two ports of two LAEs, only if both LAEs are children of the same `GroupElement` father.
2) A `Channel` can connect two ports of two LAEs, only if both ports of the two LAEs have a reference to the same `VariableGroup`.
3) A `Channel` connects exactly a `IN` port, `OUT` port pair with each other.
4) Ports of the outer `GroupElement` must be connected to ports of the inner LAEs.
5) A `Channel` connects exactly two ports.
6) A `Channel` must have an association `referencedSource` to the `OUT` port and must have an association `referencedTarget` to the `IN` port.
7) A `VariableGroup` can only be associated by exactly one `OUT` port.
8) Only an `AbstractPort` of `MultiplicityPort` type `MultiOUTChannel` can be associated by more then one `referencedSource`.

On the left side of Figure 4 a further developed block diagram of Figure 3 is shown. The logical architecture in the diagram violates rules 1, 2, 4. The diagram elements that violate a rule are shown in red. For example, the port of `AtomicElm3` in `GroupElement Elm1` is directly connected by a channel to the port of the inner `AtomicElm3` of `GroupElement Elm2`. This is a violation against rule 1. Rule 4 is also violated, because `Elm2` has an `OUT` port that is unconnected. Moreover, at the block `AtomicElm3` two ports are connected that have associations to different `VariableGroups`.

Implement layer rules: The EMAB description of the module architecture is valid, if the following rules are fulfilled:

1) An export variable of an outer MAG must have an appropriate export variable of a inner MAE. Both MAE have associations to the same variable declaration.
2) A MAE may declare a `Variable`, if it is a `Module` or an `IntegratedArchitectureElement`.
3) A MAE may have the association `import` to a `Variable v1`, when there already exists a `Variable v1` that is associated as `export` by a MAE.
4) A `Variable` that is associated as `import` by an outer MAG, must have an inner MAE that has an `import` association to the same `Variable`.
5) A `Variable v1` can be associated as export by an MAE `m`, when the `Variable v1` is declared by the MAE `m`.

On the right side of Figure 4 an extended module architecture block diagram of Figure 3 is shown. The architecture in the diagram violates implement layer rule 1. Elements in the block diagram that violate a rule are shaped in red. For example, the outer block `G2` has an association to `Variable var2` but `G2` has no inner MAE with has the appropriate `export` interface.

Mapping rule: The EMAB description for mapping the module architecture to the logical architecture is valid, if the following rules are fulfilled:

1) A `Variable v1` may have an association `VariableMapping` to a `VariableGroup vg1`, only if the `VariableGroup vg1` and the `Variable v1` are of the same `CommunicationType`.

The block diagram of Figure 4 contains mapping elements. One of these mappings is shaped in red, because mapping rule 1 is violated. Rule 1 is violated, as `var3` of `CommunicationType State` and `var1` of `CommunicationType Data` have a different `CommunicationType` and are mapped by a connection.

In contrast to these technically originated rules, the conformance checking rules ensure the validity of some domain specific layered architectural concept (see Section V).

V. Architecture Conformance Checking

In the Section IV-C the syntax and general low-level semantic checking of an EMAB model instance is performed. This section focuses on the individual product and the corresponding product line. Conformance checking and its objectives were shortly introduced in Section III-C, now we center the technical aspects. The architectural concepts defined by the dedicated architecture and represented by its architectural rules are the input for this architecture conformance checking activity together with the implemented modules, respectively the realized parts of the systems. Output of a check is a set of violations, so a list of pointers where the implementation does not fulfill the defined architecture.

As it is known from the field of architectural concepts and design patterns, these can be defined generally but it is not

uncommon, that a software architecture contains more than one concept or pattern. Thereby the patterns can be adapted or modified to meet special requirements in a different level of specialization. Furthermore, the number of concepts and its variations are increasing stately in practice. On the other hand, it might happen that realized concepts are dropped during the ongoing evolution steps. Hence, this activity has to be managed and supported by tools to support architects and developers, and for making concepts explicit.

In the following subsections the fundamentals are described individually and explained with the help of a simple example. Next, the individual checking activities, which are part of our methodology, are illustrated in detail.

A. The "Checking" Views

As explained in Section IV and shown in Figure 3, we provide different views towards visualizing the architecture of a product and a product line. LAEs have to be referred to implementation artifacts for product development. Therefore, the EMAB meta model determines that the MAE can reference at most one LAE using the `mapArchitectureElement` to determine the appropriate module elements of a logical element.

The DESIGN layer focuses on the logical architecture. In the block diagram in Figure 5 a very simple example is given with three instances of a LAE, two `AtomicElements` and one `GroupElement`. Whereby the `LAE3` declares the `VariableGroup X` and references it by an AEP as an `OUT` port type. The `LAE2` defines an AEP as an `IN` port type, which references the same `VariableGroup X`. Nevertheless, a `Channel` is specified within our simple example connecting this two ports. This is the typical view, in which a high-level architecture is specified on the PLA as well as on the PA level.

Figure 5. DESIGN view as part of the meta model instance

The IMPLEMENT view shown in Figure 6 represents the MAEs instances as blocks and their connection as concrete connections. Moreover, each MAE is referencing one LAE visualized by dashed connections between a module element block and logical element block. Regarding the development process this is, e.g., the typical view a concrete realization of a product is developed.

One simple conformance checking activity is the review of valid connections between MAEs towards the constraints specialized by the product line architecture. During development it might happen that a communication between two elements is implemented, which is not allowed with respect to the logical architecture. Because of the mapping between LAE and MAE this violations can be detected automatically and visualized.

Figure 7 shows the CHECK view for our simple example, checking the conformance rule on connections of the DESIGN layer's logical architecture elements and IMPLEMENT layer's module architecture elements. In this case the rule can be specified by an *object constraint language* (OCL) rule. This OCL rule is drawn at the top of Figure 7 and is fulfilled as

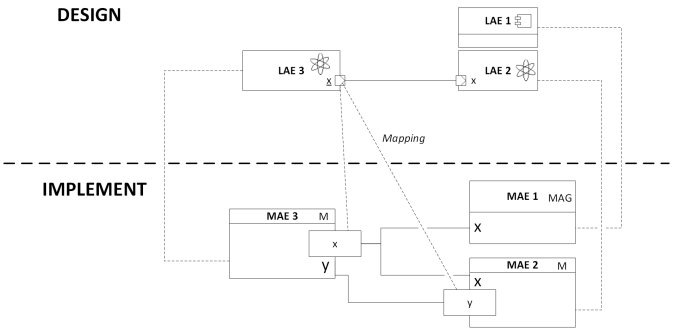

Figure 6. IMPLEMENT view as part of the meta model instance

the specified connection between the two module architecture elements corresponds to the connection between the mapped logical architecture elements.

As illustrated in Figure 7 only one of the three existing communication channels existing in the realization is permitted. Two of them (marked with the red X) are architecture violations. The connection between `MAE3` and `MAE1` is not allowed because of the mapping to `LAE3`, respectively `LAE1`, this communication is not specified. Therefore it is not valid to establish a communication between these elements in a concrete implementation.

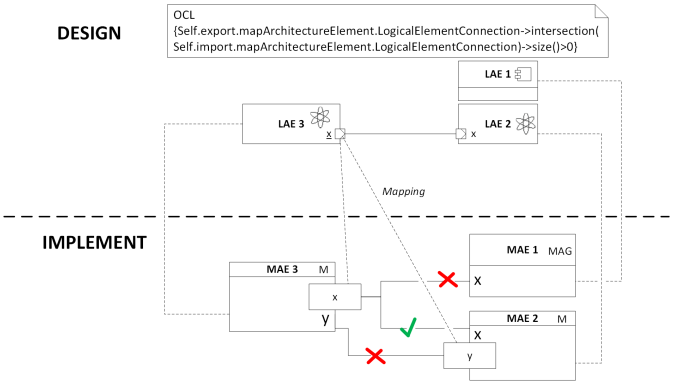

Figure 7. CHECK view as part of the meta model instance

The violation between `MAE3` and `MAE2` results of how the communication between these elements is established over variable `Y`. In the design it is specified that the variable is exported by `LAE3`, which is mapped to `MAE3` and this element is importing and not exporting variable `Y`. Hence, this results in a violation, as the realization is not conform to the specified architecture.

This simple example illustrates how the checking activity is performed in general. In the next subsections this general approach is discussed in detail corresponding to the concrete activities defined by our overall development cycle.

B. The "Checking" Activities

In this subsection the specific checking activities are described in detail. The checking takes place between a DESIGN and an IMPLEMENT instance in general and is performed on a model-to-model checking level due to the high degree of code generation from models in the automotive domain.

As illustrated in Figure 2 we defined two explicit checking activities, one on the product line level, and one on the product level.

1) Activity PL-Check: This activity contains of two scenarios, first the conformance checking of the realized elements on the product line level, these are elements, which are realized for reuse in nearly every product. Second, the holistic conformance checking taking the PLA and all products, which are existing and derived from this product line architecture, into account.

The first case is equivalent to the checking activity on product level and is described in detail in the following subsection. In the other case special challenges exit contingent on the characteristics of a product line itself. As outlined a product line architecture specifies a set of concepts, which are represented as architectural rules. The goal of the holistic conformance checking on PLA level is to check if all existing products are fulfilling these requirements. What leads to violations on this level? - Because of the high number of parallel existing product variants, which are developed individually from different teams with different know-how available using development paradigms. Nevertheless, each product has its own additional requirements. Theses are reasons why given presets by the PLA architecture are violated during development.

But also the planned extensions can lead to inconsistencies. In these cases it is very important to make the effects of architectural changes visible to get a feeling of it and to act in an adequate way. A simple example for this are the different versions of modules existing in parallel. Each architecture element of the product line development and of the product development is kept in a repository, which provides a version control capability. Predecessor relations are defined in case of modifications of an existing version. The repository also enables the selection of elements for product line development or product development. As defined in the meta model in Figure 3 an `AtomicElement` can be realized by more than one `Module`, e.g., by the same `Module` in different versions, respectively. This can lead to architectural violations if the architecture of a MAE changes during development and older versions of the same MAE are in use in other products. So as a result, a module is only applicable for a particular architecture in a concrete version. On the other hand, if the logical architecture is changing, than it could happen that not all versions of a MAE meet the requirements. The checking activity makes this visible by the set of violations.

2) Activity P-Check: Referring to our development cycle (see Figure 2) the realization of new functions or even a new product is triggered by activity `PL to P`, which defines a project plan by selecting a concrete project and set of MAEs. In addition, a concrete product can have some special requirements, especially when we have a prototype realization with the aim to evaluate a new complex function. Such requirements, which are inserted by activity `P-Requirement`, are considered in the `Design` activity and realized by `P-Implement`.

The aim of the activity `Check` is to check whether all the specifications of the product line architecture have been adhered and if the additional developments at product level fulfill the requirements of the planned product architecture.

During development, it might happen that some specifications defined by the product line and the product architecture are violated, maybe the architecture is not realizable due to technical reasons, for example. To prevent architecture erosion a checking step as introduced in subsection V-A is performed on the product level during the development cycles.

Input is the concrete realization of a product and further developed architecture of the concrete product. Specialized or additional added architecture rules are derived from this architecture. If an inconsistency between implementation and architecture exists, there are two ways to deal with these violations: Modifying the architecture or modifying the realization. It is the aim of the checking activity to make violations visible and to support this decision making process. If the developer was perhaps not familiar with the architecture for example, and this is the reason for the violations, it can still be fixed at the product level so that no erosion can occur.

However, if it is decided that there will be an adaptation to the architecture, this may have a major impact, as the concepts of the product line architecture may no longer be fulfilled. On the other hand, this also offers the opportunity to transfer evaluated and tested modifications into the product line to roll them out to all or nearly all other products.

3) Effects of the Checking Activities: The potential effects of the checking activities to other activities are visualized in Figure 8 (cases 1-3, red arrows). In every case the effects belong to the detected violations between architecture and realization and to eliminate them the architecture and/or implementation has to be adapted.

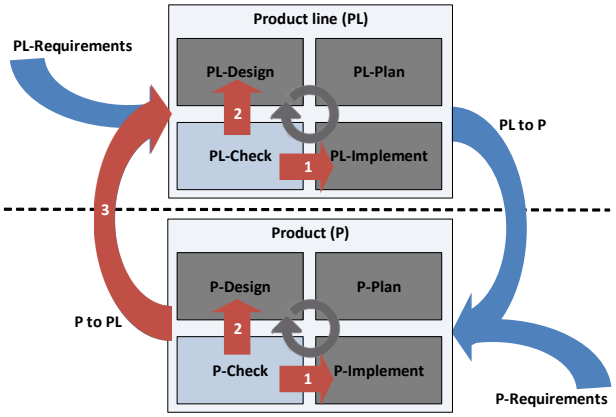

Figure 8. Potential effects of the checking activities

Case (1) describes the adaption of the implementation, this effects the `Implement` activities both on product line (`PL-Implement`) and product (`P-Implement`) level. This case arises if the detected violation is declared as an implementation fault and the architecture should be realized as specified.

Case (2) effects the `Design` activities as well on product line (`PL-Design`) as on product (`P-Design`) level. The detected violation results in an adaption of the architecture not the implementation. This typically occurs if an architectural constraint is not realizable on technical level respectively it was realized in a “better” way, so it was decided to bring the implemented concept to the architecture level.

Case (3) is similar to case (2) but can effect all products of a product line. As mention it is very common to elaborate new functionalities especially complex functions, which might influence the software architecture. After testing, evaluating and approving it in a single prototype product, the violations on the product level can be resolved by case (1) or (2). But if there are major changes, it might happen that the architecture of the prototype does not fulfill all requirements of the product line architecture. In this case the effects of these changes have to be discussed on product line level. This can result in an adaption of the whole product line, if it seems reasonable to integrate the new developed concepts of the prototype in the product line architecture to improve the existing products. In the other case, if it is not reasonable to modify the product line architecture, the product that was originally derived from the product line will be marked as a special product variant. So deviation of a product architecture from a product line architecture is handled in a managed way.

The results from the checking activity support the synchronization between product and product line, because it is pointed out, where the product architectures differ from the guidelines defined by the product line architecture. Therefore, we have to keep in mind that in practice more than one product variant is in development: there are many that are developed in parallel, which leads to architectural changes, and, without syncing these variants in a managed process, the changes will force the erosion of the product architecture as well as the product line architecture.

C. The "Checking" Tool Support

Due to the high number of product lines existing in parallel and the high number of derived and realized product variants it is not economical to implement such a checking activity without a suitable tool support. Thereby the following use cases have to be supported:

1) Creation of architecture rules,
2) selecting the implemented artifacts,
3) selecting the right rules,
4) performing the check and
5) visualizing the checking results to determine further actions.

For performing the checking activity we use an approach based on a logical fact base, as described in [13], [31].

In Figure 9 the checking process is visualized. First the implemented artifacts that should be checked, have to be selected as module architecture instance. This model instance is transformed to a logical representation. In parallel, the architect derives and/or selects the architectural rules, which are representing the architectural concepts specified by the corresponding logical architecture instance. Today this rule creation step is a human centered process and not automated.

After transferring the software artifacts into a logical fact base and describing the architectural concepts by first-order logic statements, queries can be executed on the fact base to validate the rules. This results in a list of violations or in the best case the implementation fulfills all architecture rules, so the implementation is conform to the specified logical architecture.

Now it is the task of the developer respectively the software architect to handle the violations. This is similar with the rule creation, the second step, which is performed and based on the experience of a human to resolve possible architectural violations.

We define the *Checking* activity as a separate activity for two reasons. On the one hand, it is a process composed of several steps and, on the other hand, the checking is a snapshot based step. So, related to a development process, it makes sense to have an explicit activity triggered by launching new snapshots of a system.

Focusing the tool support, an important open issue is the creation of architectural rules, but also the extraction of architectural concepts introduced by the developers best practice and making them explicit. Consequently, we have some ongoing research in this field. One approach is to concentrate on the implemented artifacts and extracting architectural concepts from it with the help of machine learning techniques to compare them with the given logical architecture to support the decision making process described in the previous subsection.

VI. Real World Example: Brake Servo Unit (BSU)

In this section, we present an example of a software system we developed in cooperation with Volkswagen. The main task of this system is to ensure a sufficient vacuum within the brake booster that is needed to amplify the driver's braking force. At first, we describe the context the system is embedded in and a view onto the system's structure. We show how the system has evolved. After the presentation of the mapping of the evolution onto our approach, we give results and a discussion.

A. System Structure and Context of BSU

In vehicles, a vacuum brake booster (brake servo unit/BSU) is mounted between the brake pedal and the hydraulic brake cylinder. It consists of two chambers separated through a movable diaphragm. If the driver is not braking, the air is evacuated from both chambers. When he pushes on the brake pedal a valve opens and atmospheric pressure air flows into one chamber. Due to the differential air pressure within the BSU the diaphragm starts to move towards the vacuum chamber creating a force. This force is used to amplify the driver's braking force.

The vacuum can be generated using different techniques. The BSU is either attached to the intake manifold using its internal lower pressure or to an electrically or mechanically driven vacuum pump. Using the intake manifold as vacuum generator can be problematic. Special operating modes of other vehicle's subsystems can increase the intake manifold pressure so much that its internal vacuum is not sufficient to evacuate the BSU when needed.

The software system realizes a set of feedback controllers to reduce the disturbances caused by other systems or to switch on the vacuum pump, respectively. Since it makes no sense to use all controllers at the same time it is necessary to coordinate their activation. Besides the controlling of BSU vacuum and the coordination of controllers, the software has to provide valid pressure information all the time. For this purpose the software selects from several sensors the one that provides the best quality of pressure information. The logical view of the designed architecture is presented in Figure 10.

The BSU hardware system is part of a wide range of products within the huge family of cars. Since the diversity of

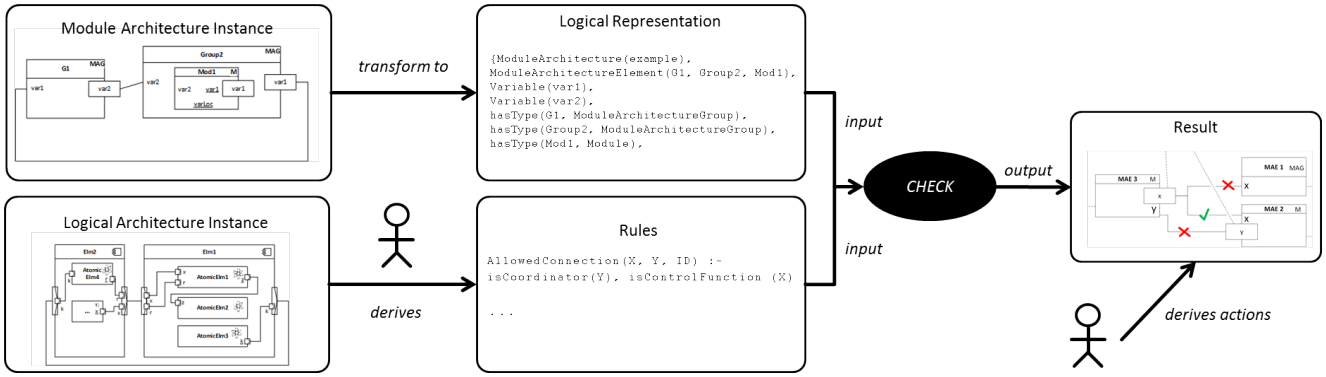

Figure 9. Tool supported checking process

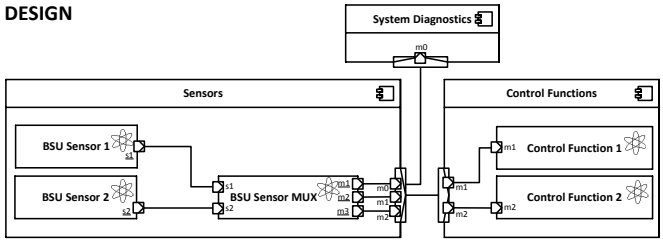

Figure 10. Logical view of the software architecture of BSU

the used hardware components like sensors and actuators that are mounted to the braking system and features that influence the BSU software one important goal of the architecture development was to support variability. The BSU software system is decomposed into two major parts: `Sensors` and `Control Functions`. The decomposition of the `Sensors` component into parts for every sensor type each allows a one to one mapping from features to components. To realize variability in an efficient way, standardized interfaces are used for communication. A coordinating component `BSU Sensor MUX` just has to provide a sufficient amount of ports for the interaction with the sensors and control functions.

The `Control Functions` component is decomposed using a similar technique. Every control function is realized by a specific component. These components provide standardized interfaces for communication with subsequent vehicle functions, which must follow the BSU commands, e.g., disable the start-stop system (not depicted in Figure 10).

B. Evolution of BSU

As it was customary in the automotive domain, BSU's hardware and software have been implemented by various suppliers in the past. The requirements for the functionalities of the system were the same for all suppliers, but there were differences in the type of implementation by the respective suppliers. During the further development of the system over many years, new requirements had to be continuously implemented. Examples of this are the support of various engine variants such as otto, diesel and electric engines. As the range of functions increased, the essential complexity grew; however, the accidental complexity [32] has increased disproportionately. The growth of accidental complexity results from a "bad" architecture with strong coupling and a low cohesion, which have evolved over the time. Despite extensive further development of the system, the original structure of the software was not adequately adapted. Overall, the monolithic structure of the software remained. The software consisted of a single software module, which, however, was internally characterized by increasing accidental complexity. The variability was realized completely by annotations. Thereby, the system's maintainability and expandability has been complicated additionally.

In recent years, many automotive manufacturers have begun to develop software primarily in-house to save costs and to secure important know-how. However, the hardware components are still being developed by the supplier companies in general. Against this background, Volkswagen decided to develop the BSU in-house in the future. Together with our institute, Volkswagen developed its own software for the BSU in 2012 on the basis of the existing system. Configurability, extensibility and comprehensibility were defined as essential quality targets. In addition, new architecture and design concepts have been introduced to meet these quality objectives in the long term and permanently.

After successful introduction of the system into series production, the software system was continuously developed after 2012. In all, the BSU system was reused in more than 140 project versions, some of them with adaptations. There were, e.g., the introduction of five additional control functions that were necessary because of changes to the system environment. This includes, in particular, the introduction of new components such as actuators, which were essentially driven by the electrification of the powertrain. In the following sections, we will present our methodology by means of the BSU's further development and discuss the results. However, due to the obligation of secrecy, we can not name real-world functions. Instead, we will abstract from real control functions, actuators, and sensors in the following sections.

C. Application of our Approach to BSU Further Development

In this section, we will outline the evolution of BSU further development, described in the previous section, mapped to the overall development cycle visualized in Figure 2. As mentioned in Section VI-B, the development started in 2012 and continues until today. We will pick out the milestones of this evolution process and explain in detail, how our approach supports the management of development. Therefore, we will describe the further development of the BSU chronologically.

The architecture of the BSU at this point is equivalent to Figure 10.

The first considerable development activities leading to architectural evolution results from two new control functions. These new control functions are specified as product line requirements (PL-Requirement). In the following activity PL-Design, the new requirements including all open requirements and feedbacks from the ongoing product development activities submitted by activity P to PL, are took into account by the designing of the new PLA (called "PLA vision"). The resulting PLA includes two new components, whereby each component represents one of the new control functions.

After assessing and determining the new PLA vision, the PL-Plan activity starts. It was decided to realize the new PLA vision in two iterations, per iteration cycle, one of the new components should be implemented completely. Regarding to the development plan in activity PL-Implement the first component was implemented. PL-Check activity is triggered after the new component is fully implemented. In this activity, the conformance of the implementation is checked against the planned architecture (PLA vision), as illustrated in Section IV. An example for an architectural rule, which was checked, is defined informally as follows: "Each BSU Sensor has to communicate to Control Functions by the BSU Sensor MUX". This rule can be easily derived from the logical view shown in Figure 10. The outcome of the checks was positive so the next iteration was started.

Parallel to the implementation of the second defined component some concrete products are selected to integrate the new developed control function in real products (activity PL to P). It was decided to setup a new pilot product additionally. The pilot got a special requirement by a P-Requirement activity. The proving by prototypes or pilots is a common approach in the automotive domain. Due to the specification of the special requirement, which includes a new control function with a coordinating feature, a prototyping approach was used to realize this requirement. This simply means that we have a main control function and a backup control function, if the main function is not available the backup function should be used.

The solution of the P-Design activity was a solution that fulfills all requirements. It was decided to add a new component representing the new control function and to establish an additional coordinator component. The coordinator has the responsibility of the controlling of main and backup functions and realizing the coordinating feature.

In the P-Plan activity the iterations to be performed had to be defined and scheduled. The outcome was a development plan with two iteration steps. In the first step, the new control function and the coordinator component should be implemented. And in a second step, all existing control functions had to be adapted, because they had to be defeatable to perform as main or backup function.

According to the development plan, the P-Implement activity was performed. After each iteration step, a conformance check was done (P-Check). In our case study we detected a violation of an architectural rule. Consequently, it was evaluated and discussed, if the solution of the violation results in adapting the implementation or in adapting the architectural rule itself - or in simple words, is there a crummy implementation or an insufficient architecture (cf. Section V-B). In terms of internal classification we cannot go in detail at this point.

After evaluating the product realization all adaptations and changes of architecture and implementation are forwarded to the product line architecture level by a P to PL activity. These are inputs for the next PL-Design activity, thereby it had to be decided, which changes should be integrated into the product line architecture and its implementation or otherwise, which had to be declared as a "special" solution. In our case the coordinator concept was established in the product line. This leads, e.g., to the new architectural rule: "Each Control Function must have a communication connection to the Coordinator", which is now mandatory for all products derived from this product line. The final architecture is visualized in Figure 11 including all newly developed control functions, the Coordinator component, and the additional connections between the control functions and the Coordinator for the controlling of activation.

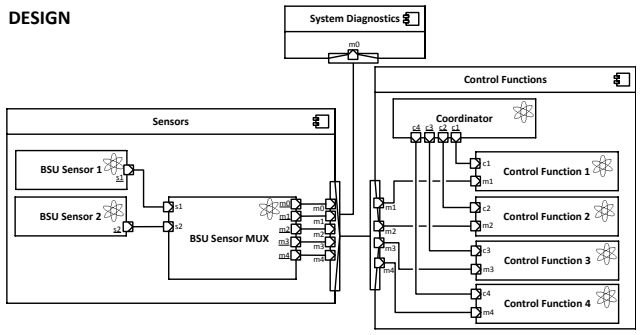

Figure 11. Logical view of the software architecture of BSU including the coordinator concept and the three new control functions (in 2012)

In summary, the architecture of the BSU is largely stable after the introduction of the coordinator concept until today.

Overall we state that our approach can deal with many parallel activities at product line and product level. This becomes apparent by the controlling character of the synchronization points both in the development cycle on product line and product level by activities PL-Check and P-Check and between the product line and product level by activities PL to P and P to PL. In this way, it was possible to detect architecture erosion in an early state and to take adequate countermeasures. Furthermore, we can take care of a planned generalization on the one hand and a planned specialization or exceptional case handling on the other hand. This is evidenced by the coordinator concept: A concept designed and fully realized and proved by a pilot product and than transferred into the product line architecture and finally fully integrated within the next development iterations in the product line architecture and all products belonging to this architecture.

VII. Evaluation and Discussion

In this section we present and discuss the results of a field study of a five year BSU software development and evaluate, if the main objectives introduced in Section I are fulfilled.

A. General Results of the Quantitative Analysis

To evaluate our methodology, we present the quantitative analysis for the BSU software development that is realized

TABLE II. RESULT OF THE QUANTITATIVE ANALYSIS FOR THE BSU SOFTWARE FOR THE INTERVAL OF 5 YEARS.

	Count	*Number of versions*	*Average number of versions*	*Min. number of versions*	*Max. number of versions*
LAE	15	15	15/15 = 1	1	1
MAE	15	57	57/15 $\approx$ 4	1	6
Projects	21	146	146/21 $\approx$ 7	1	12

TABLE III. FURTHER RESULTS OF THE QUANTITATIVE ANALYSIS FOR THE BSU SOFTWARE.

	Number of versions used in projects	*Cumulated number of versions used over all project versions*	*Average degree of reuse of each version*	*Number of used design configurations*
MAE	46	1611	1611/46 $\approx$ 35	n/a
Projects	n/a	n/a	n/a	14

and maintained in cooperation with our project partner over a period of 5 years. In the following, we focus on the applicability of the product line and product development activities. Two criteria are important to evaluate. First, the amount and kinds of modifications on architecture elements calling this *complexity controlling*. Second, the amount and kinds of design configurations calling this *variant controlling*.

Table II shows the result of the quantitative analysis. The data record for the quantitative analysis refers to the development of the BSU software and the product realizations consisting of the BSU software and further vehicle functions. The record contains the version control graph of the past 5 years of BSU software development, called *repository* in the following. Each node is a version of an architecture element or realized product. Edges connect two subsequent versions. Table II shows the number of logical architecture element (LAE) versions, of module architecture element (MAE) versions, and of project versions from the record. Modifications were triggered by the realization of BSU software `PL-Requirements` or by the realization of products due to `P-Requirements`.

Table II shows the count of 15 LAE referring to modifications at the DESIGN layer and the count of 15 MAE referring to modifications at the IMPLEMENT layer. The kind of modifications refers to the connection structure and to the architecture element structure of the appropriate MAE. Each LAE is available in exactly one version in the repository. Thereby, the current state of the logical architecture is represented, which is unmodified since the beginning of the record. Unfortunately, the data of prior development stages of the BSU software logical architecture is not considered by the record due to data protection reasons. In total, 57 versions for MAE exist. A module element of the module architecture was modified in minimum 1 time, in maximum 6 times, and in average 4 times. Thereby, each version of the MAE is mapped in this case to exactly one version of the appropriate LAE.

Line "Projects" in Table II refers to the product development of the BSU software and shows that 21 projects containing the BSU software exist. A project defines a set of architecture element versions from logical architecture and from module architecture used to realize a product. In the following, we call the set of versions of architecture elements *design configuration*. Each time a project is modified, a new version of that project is committed to be used for subsequently realize the product. The project modifications resulting in a new version commit always refers to changes of the design configuration. In total, the project version number is 146. The average number of versions is 7, the minimum number is 1, and the maximum number is 12.

The data in Table III shows two quantitative aspects. First, the number of BSU software architecture element versions used in projects is 46 and the cumulated number of BSU software architecture element versions used in all project versions is 1611. Hence, the average degree of reuse of each version of MAE is 35. Second, the number of different design configurations of all project version concerning the BSU software is 14. This induces the fact that 14 architecture structure variants of the BSU software architecture (logical and module) are used in projects to realize products in the past 5 years.

Complexity controlling: Complexity in BSU software is induced by modifications on architecture elements of the logical architecture and the module architecture, which are triggered to realize the two kinds of requirements described by the record. To handle complexity, each modification must be controlled for violations on architecture elements and on violations referring quality properties.

Our methodology aims to control violations of quality properties in the `Design` activity and of violations of architecture rules in the `Check` activity. The `Design` activity provides the modified DESIGN layer in each iteration and the `Implement` activity provides the modified IMPLEMENT layer in each iteration. The BSU software modifications are applied to realize requirements resulting in a product dependent BSU software or in a new product independent realization of the BSU software. Therefore, `PL-Requirements` corresponding to new features triggers the controlling of BSU software modifications during the product line development activities, using the versions of logical architecture at the DESIGN layer and of versions of module architecture at the IMPLEMENT layer. New project related requirements corresponding to `P-Requirements` triggers the product development activities to control all modifications considering project related versions and architecture related versions corresponding to the appropriate layers and of the EMAB meta model.

After applying the methodology two important results are observed: First, no violations on architecture quality properties at the DESIGN layer were found. Second, after checking the modifications of the BSU software applying inter alia the rule described by the EMAB meta model in Section IV, only one minor violation between the layers of the BSU software was found. This evaluation result shows that nearly all modifications of BSU software in the past 5 years preserved the architecture conformance of the IMPLEMENT layer to the DESIGN layer. Moreover, the structure of the DESIGN layer is well realized considering the quality properties.

Variant controlling: The term variant in the case of BSU software describes a software architecture variant reused to realize a software product. Thereby, each project version refers to exactly one design configuration to define architecture elements for reuse that are contained in the software architecture variant. Modifications of the logical and module

architecture can introduce violations on expected derivable structure variants. To handle such violations the control of variants must be applied to the modifications. The control of such architecture rule violations is applied during the `Check` activity of the product line development considering the versions corresponding to the IMPLEMENT layer and to the DESIGN layer. After applying our methodology, no violations are found in the past 5 years of development. This corresponds to the result of complexity evaluation where conformance of the EMAB layers is confirmed.

B. Evaluation with Regard to the Main Objectives of our Approach

We will evaluate, if the main objectives introduced in Section I are fulfilled by the BSU example. These objectives are:

1) Maintaining stability of the PLA and minimizing product architecture erosion even if extensive further development of the system takes place.
2) Achieving high scalability, and a high degree of usage of the modules.

The first objective focuses on the software architecture issues whereas the second objective regards the software components. Obviously, the following four sub-objectives can be derived by the two major objects:

1) Stability of a product line architecture
2) Erosion indicator
3) Usage of a module
4) Scalability of a module

1) Definitions of Evaluation Criteria: We will explain each sub-objective in detail and by proposing metrics to evaluate the sub-objectives.

General notations:

- $PLA_t :=$ active PLA at point t
- $P_{t,x} :=$ project variant at point t, which is corresponding to the PLA_t with the same t. This is feasible because only one PLA is active at a point t in the case study. No point t exists where two PLAs are active.
- $p_{t,x} :=$ an explicit version of a project $P_{t,y}$ at point t.
- $|P_{t,x}| :=$ number of explicit versions of a project variant $P_{t,x}$ at point t.
- $M_{t,x} :=$ module / realization artifact at point t, which is corresponding to the PLA_t with the same t.
- $m_{t,x} :=$ an explicit version of a module $M_{t,y}$ at point t.
- $|M_{t,x}| :=$ number of explicit versions of a module $M_{t,x}$ at point t.
- $\Phi_{PLA_t,P} := \{P_{t,1}, P_{t,2}, ..., P_{t,n}$, is defined as the set of all projects existing at point t and derived from the given PLA.
- $\Phi_{PLA_t,M} := \{M_{t,1}, M_{t,2}, ..., M_{t,m}\}$, is defined as the set of all modules existing at point t and derived from the given PLA.
- $\Phi_{PLA_t} := \Phi_{PLA_t,P} \cup \Phi_{PLA_t,M} = \{P_{t,1}, P_{t,2}, ..., P_{t,n}, M_{t,1}, M_{t,2}, ..., M_{t,m}\}$, is defined as the set of all projects and modules existing at point t and derived from the given PLA.

Stability of a product line architecture:
is described by the amount of changes between two PLAs. The stability $Stab$ and the number of project versions existing during the period a PLA is active, NAP, are defined as follows:

$$Stab(PLA_t) = \frac{w(PLA_{\Delta t-1,t})}{\frac{w(PLA_{t-1})+w(PLA_t)}{2}} \tag{1}$$

with weight $w(PLA_t) = |N_{PLA_t}| + |E_{PLA_t}|$, whereby $N_{PLA_t} :=$ set of nodes of PLA_t and $E_{PLA_t} :=$ set if edges of PLA_t.

The delta-graph $w(PLA_{\Delta t-1,t})$, which is containing the changed elements, is defined by the edges that are added or deleted comparing PLA_{t-1} and the following PLA_t and all nodes connected by these edges.

In addition $Stab\,(PLA_t) \rightarrow [0, 2(|N_{PLA_t}| + 1)]$, whereby a high value indicates comprehensive changes and a low value small diversities. The assumption is that the value should not be higher than 1, but this had to be validated.

$$NAP(PLA_t) = \sum_{x=1}^{|\Phi_{PLA_t,P}|} |P_{t,x}| \tag{2}$$

Erosion indicator
The erosion of a PLA is indicated by the number of violations of project versions. Therefore the mean number of violations MV is defined as,

$$MV\,(PLA_t, P_t) = \frac{v(PLA_t, P_t)}{|P_{t,x}|} \tag{3}$$

with

$$v(PLA_t, P_t) = \sum_{x=1}^{|P_{t,x}|} v(PLA_t, p_{t,x}) \tag{4}$$

and $v(PLA_t, p_{t,x})$ is defined as the number of violations of $p_{t,x}$ in PLA_t.

Usage of a module
The usage of a Module $M_{t,x}$ is defined as how many project variants P_t at point t use a version of $M_{t,x}$. Therefore we define τ as,

$$\tau(p_{t,x}, m_{t,y}) = \begin{cases} 1 & \text{, iff project } p_{t,x} \text{ uses module version } m_{t,y} \\ 0 & \text{, otherwise} \end{cases} \tag{5}$$

The same applies for $\tau(P_{t,x}, M_{t,y})$, in this case it means that any version of a Module M_t is used in any version of a project P_t.

Furthermore we define τ for a set of projects of a PLA as,

$$\tau(\Phi_{PLA_{t,P}}, m_{t,y}) = \begin{cases} 1 & \text{, iff a } p_{t,x} \text{ exists that uses } m_{t,y} \\ 0 & \text{, otherwise} \end{cases} \tag{6}$$

So the usage u is defined as,

$$u(\Phi_{PLA_{t,P}}, M_t) = \frac{\sum_{x=1}^{|\Phi_{PLA_{t,P}}|} \tau(P_{t,x}, M_t)}{|\Phi_{PLA_t,P}|} \quad (7)$$

In addition $u(\Phi_{PLA_{t,P}}, M_t) \rightarrow [0, 1]$, where a value near one means a high penetration of module M_t and a low value corresponds to a minor usage of the module.

Scalability of a module

The scalability $Scal$of a module describes the parallel usage of different versions of a module M_t and it is defined as,

$$Scal(\Phi_{PLA_{t,P}}, M_t) = \sum_{y=1}^{|M_{t,x}|} \tau(\Phi_{PLA_{t,P}}, m_{t,y}) \quad (8)$$

The value for the scalability of a module should be one. It is not profitable have a high value, because this means that the variability of a module is realized by different versions of a module, which results in a high maintainability effort. As well as a scalability value of zero is not preferable, because this indicates the non-use of a module.

2) Evaluation of the Defined Criteria: Next we apply our metrics to the BSU example. The discrete timestamps for t are the following five values $t = \{2012, 2013, 2014, 2015, 2016\}$ for the year 2012-2016.

Table IV shows the evolution of the BSU over the period 2012-2016 and is the basis for the following evaluations. Column *MAE vers.* contains the version number of the MAE. Version "1" is the first version of a MAE indicating a new development of a MAE. The total number of MAE versions is 57.

Stability of a product line architecture:

For the calculation of stability we consider the PLA of 2012 and 2013. The PLA for the year 2013 is shown in Figure 12. Figure 11 in the previous section shows the PLA for 2012. In order to calculate the stability, we have constructed a simple graph with the nodes and edges for both PLAs in Figure 13. Hierarchical components such as `Sensors` are also represented by nodes. Channels become edges in the graph. Multiple channels between two nodes are combined into one edge.

The graph for the PLA from 2012 is depicted on the left side in Figure 13. The weight w is calculated from the sum of nodes and edges and is 24 for PLA_{2012}. On the right side the graph for the PLA from 2013 is shown. There is a new node, `Control Function 5`, and two additional connections. However, there are also adjustments of the channels up to the `BSU Sensor MUX`. All changes are indicated by the dashed lines. In the middle of Figure 13, the delta graph is shown. All added and modified edges with the associated nodes are included there.

Below we calculate the stability for the years 2013-2016. Since the stability always calculates the change to the previous version, there is no calculation for 2012. As there was no further development of the PLA in 2014 and 2016, the stability is 0 in both years. Even in the years with adaptations of the PLA, a relatively small change effort results, which leads to a low weight of the delta-graph. Overall, a high stability of the PLA can be observed over the period.

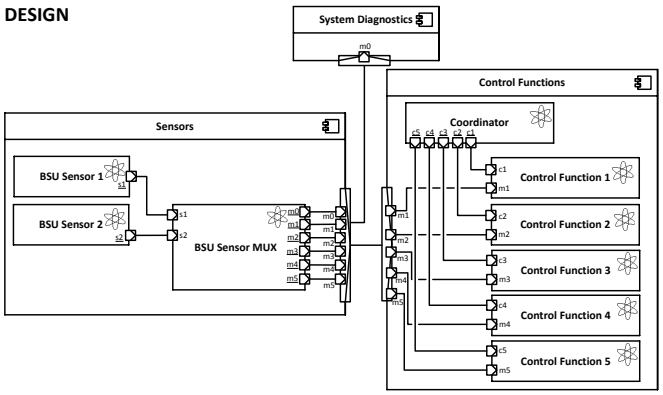

Figure 12. PLA of the BSU in 2013

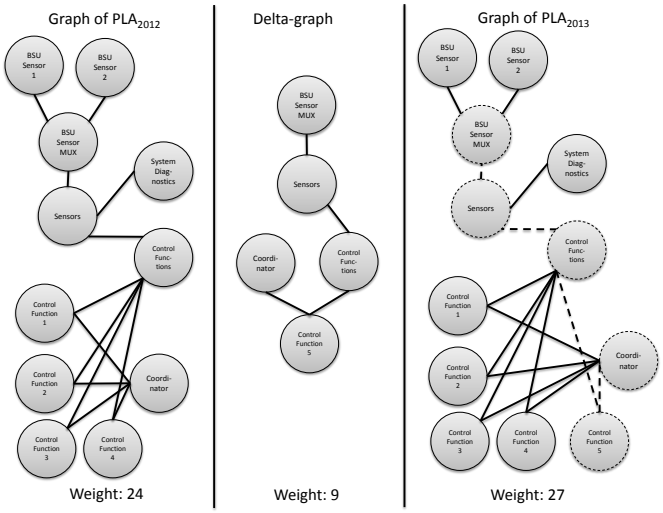

Figure 13. Building the delta-graph (graph in the middle) of PLA_{2012} (left) and $PLA_{2013}(right)$

$$Stab(PLA_{2013}) = \frac{w(PLA_{\Delta 2012,2013})}{\frac{w(PLA_{2012})+w(PLA_{2013})}{2}} = \frac{9}{\frac{24+27}{2}} \approx 0,35$$

$$Stab(PLA_{2014}) = \frac{w(PLA_{\Delta 2013,2014})}{\frac{w(PLA_{2013})+w(PLA_{2014})}{2}} = \frac{0}{\frac{27+27}{2}} = 0$$

$$Stab(PLA_{2015}) = \frac{w(PLA_{\Delta 2014,2015})}{\frac{w(PLA_{2014})+w(PLA_{2015})}{2}} = \frac{15}{\frac{27+36}{2}} \approx 0,48$$

$$Stab(PLA_{2016}) = \frac{w(PLA_{\Delta 2015,2016})}{\frac{w(PLA_{2015})+w(PLA_{2015})}{2}} = \frac{0}{\frac{36+36}{2}} = 0$$

In order to evaluate the stability of the product line architecture, we also want to relate $Stab$ to the quantity of project versions (NAP). After counting the versions of the projects from 2012 to 2016, we get the following values:

$NAP(PLA_{2012}) = 9$, $NAP(PLA_{2013}) = 9$,
$NAP(PLA_{2014}) = 32$, $NAP(PLA_{2015}) = 58$,
$NAP(PLA_{2016}) = 38$.

The values point to a high degree of further development, especially from 2014 onwards. Nevertheless, the values for $Stab$ are relatively low so that the overall stability of the PLA can be assessed as high.

TABLE IV. EVOLUTION OF THE BSU SOFTWARE FROM 2012 TILL 2016.

2012			2013			2014			2015			2016		
MAE vers.	*MAE name*	*Creation date*	*MAE vers.*	*MAE name*	*Creation date*	*MAE vers.*	*MAE name*	*Creation date*	*MAE vers.*	*MAE name*	*Creation date*	*MAE vers.*	*MAE name*	*Creation date*
1	BSU-A	23.04.2012	2	BSU-F	22.02.2013	1	BSU-N	11.06.2014	6	BSU-A	16.01.2015	3	BSU-O	05.01.2016
1	BSU-B	24.04.2012	3	BSU-A	12.03.2013	3	BSU-D	18.06.2014	5	BSU-C	22.01.2015	6	BSU-C	25.04.2016
1	BSU-C	07.05.2012	2	BSU-D	15.03.2013	3	BSU-B	30.06.2014	4	BSU-H	11.03.2015	3	BSU-M	25.04.2016
1	BSU-D	07.05.2012	2	BSU-H	15.03.2013	3	BSU-G	30.06.2014	4	BSU-G	21.04.2015	7	BSU-C	28.04.2016
1	BSU-E	07.05.2012	2	BSU-I	15.03.2013	4	BSU-C	01.07.2014	5	BSU-H	22.04.2015	8	BSU-C	30.04.2016
1	BSU-F	07.05.2012	1	BSU-K	25.03.2013	5	BSU-A	01.07.2014	7	BSU-A	23.04.2015	9	BSU-C	11.05.2016
1	BSU-G	07.05.2012	4	BSU-A	28.08.2013	3	BSU-E	02.07.2014	1	BSU-L	26.05.2015	10	BSU-C	12.05.2016
1	BSU-H	07.05.2012				3	BSU-F	02.07.2014	1	BSU-M	27.05.2015	11	BSU-C	14.05.2016
1	BSU-I	09.05.2012				2	BSU-J	03.07.2014	8	BSU-A	02.06.2015			
1	BSU-J	10.05.2012				3	BSU-H	03.07.2014	1	BSU-O	11.09.2015			
2	BSU-E	19.07.2012				2	BSU-K	03.07.2014	2	BSU-M	05.10.2015			
2	BSU-A	10.08.2012				3	BSU-I	03.07.2014	2	BSU-O	04.12.2015			
2	BSU-B	05.09.2012				4	BSU-E	28.11.2014						
2	BSU-G	05.09.2012				3	BSU-K	08.12.2014						
2	BSU-C	11.09.2012												
3	BSU-C	26.09.2012												
Number of versions: 16			Number of versions: 7			Number of versions: 14			Number of versions: 12			Number of versions: 8		

Erosion indicator

For a given PLA, architecture rules are derived as a means to check conformity. In this context, the degree of erosion is measured by counting the number of violations of architecture rules for a PLA per year. Ideally, this value should be kept as small as possible. Only one violation was detected in 2014. Accordingly $MV\,(PLA_{2014}, P_{2014}) = 1$. In all other cases, $MV = 0$.

Usage of a module

In Table V we specify the result of the usage calculation. Some modules were developed after 2012. No value could be calculated for the corresponding fields where the module did not yet exist. Therefore, these fields are marked with n/a. It is noticeable that module `BSU-N` is never used in the period under consideration. Many fields have the value 1, which means a complete usage. For the modules created in 2015 (`BSU-L`, `BSU-M`, `BSU-O`), the usage is initially low but improves in 2016. With module `BSU-B`, the usage deteriorates over the period of time. The causes of this should be investigated.

Figure 14 illustrates the course of the usage. Modules showing the same course of the usage are summarized.

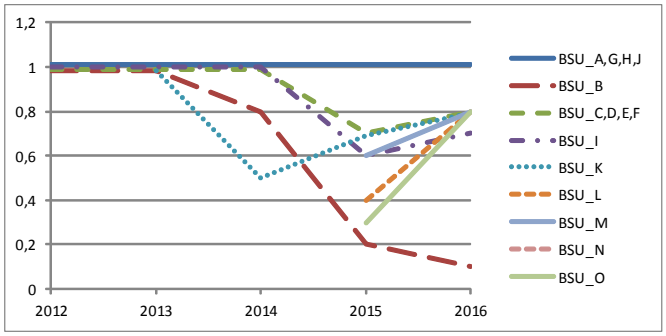

Figure 14. Usage of BSU modules A-O

TABLE V. USAGE OF A MODULE

MAE name	*2012*	*2013*	*2014*	*2015*	*2016*
BSU-A	1	1	1	1	1
BSU-B	1	1	0.8	0.2	0.1
BSU-C	1	1	1	0.7	0.8
BSU-D	1	1	1	0.7	0.8
BSU-E	1	1	1	0.7	0.8
BSU-F	1	1	1	0.7	0.8
BSU-G	1	1	1	1	1
BSU-H	1	1	1	1	1
BSU-I	1	1	1	0.6	0.7
BSU-J	1	1	1	1	1
BSU-K	n/a	1	0.5	0.7	0.8
BSU-L	n/a	n/a	n/a	0.4	0.8
BSU-M	n/a	n/a	n/a	0.6	0.8
BSU-N	0	0	0	0	0
BSU-O	n/a	n/a	n/a	0.3	0.8

Scalability of a module

In Table VI we show the calculated values of scalability. The value 1 represents the optimum value and is therefore represented by a green background color. Values greater than 1 indicate a negative scalability and are therefore shown as yellow (value = 2) or red (value $>$ 2). A value of 0 is also generally seen as negative (`BSU-N`) because a version of the module exists, but is not used in any project.

The development of the scalability of 2012-2016 is shown in Figure 15. The number of respective values is counted here. In 2014 and 2015 there is a deterioration in scalability. In 2016, the scalability of the modules improves again significantly.

TABLE VI. SCALABILITY OF A MODULE

MAE name	*2012*	*2013*	*2014*	*2015*	*2016*
BSU-A	2	3	3	4	1
BSU-B	2	1	1	2	1
BSU-C	1	2	2	2	4
BSU-D	1	2	2	2	1
BSU-E	2	1	2	2	1
BSU-F	1	2	2	2	1
BSU-G	2	1	2	3	1
BSU-H	1	2	2	4	1
BSU-I	1	2	2	2	1
BSU-J	1	1	2	2	1
BSU-K	n/a	1	2	2	1
BSU-L	n/a	n/a	n/a	1	1
BSU-M	n/a	n/a	n/a	2	3
BSU-N	0	0	0	0	0
BSU-O	n/a	n/a	n/a	2	2

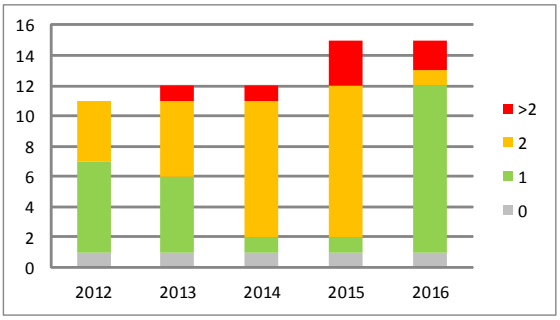

Figure 15. Number of modules with scalability 0, 1, 2, and >2

VIII. CONCLUSION AND FUTURE WORK

We proposed a holistic approach for a long-term manageable and plannable software product line architecture for automotive software systems. Our approach aims at a long-term minimization of architecture erosion, and thereby guarantee a constant high degree of reusability. Thus, we propose concepts like architecture compliance checking with specific adaptations to the automotive domain. The focus is on scalability, to manage a huge number of variants in real world automotive systems.

We introduced the description language EMAB and its meta model for the specification of the software product line architecture and the software architecture of the corresponding products. The EMAB is applied for the architecture extraction and the managed evolution approach presented in this paper.

In contrast to the validation checking of an EMAB model instance, where a syntax and general low-level semantic checking is performed, we proposed an approach for architecture conformance checking to identify architecture violations as a means to prevent architecture erosion. The architectural concepts defined by the dedicated architecture and represented by its architectural rules are the input for this architecture conformance checking activity together with the implemented modules, respectively the realized parts of the systems. Output of a check is a set of violations, so a list of pointers where the implementation does not fulfill the defined architecture.

We demonstrated our methodology on a real world case study, a brake servo unit (BSU) software system from automotive software engineering. In the case study, we could show that we have met the two main objectives defined in Section I:

1) Maintaining stability of the PLA and minimizing product architecture erosion even if extensive further development of the system takes place.
2) Achieving high scalability, and a high degree of usage of the modules.

As a result, with regard to objective (1), we could limit architecture erosion to a minimum: Only one minor violation occurred in a period of five years. All further developments have followed the originally planned architectural principles and thus resulted in a high stability of the PLA. Moreover, with regard to objective (2) we were surprised at the high number of usage of the modules: Most modules were used in all projects existing at that time. Only the scalability deteriorated in 2014 and 2015. But in 2016, the value has improved considerably again.

As a further observation a high degree of reuse could be observed: Each module was reused on average in 35 projects. Even the high number of potential variants could be managed with the approach.

As a future work, we aim at realizing a tool-chain enabling the architecture description of the different architectures (PLA, PA, including versioning), the measure and evaluation of quality attributes, as well as the integration of the ArCh-Framework [13]. Appropriate abstraction techniques are crucial to cope with the huge set of adjustable parameters within the ECU software and to manage variability. Thus, we are currently developing a concept including a prototypical tool environment that enables the description and visualization of variability.

REFERENCES

[1] C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures," in Special Track: Managed Adaptive Automotive Product Line Development (MAAPL), along with ADAPTIVE 2017. IARIA XPS Press, 2017, pp. 43–52.

[2] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of Software in Distributed Embedded Automotive Systems," in Proceedings of the 4th ACM international conference on Embedded software. ACM, 2004, pp. 203–210.

[3] A. Rausch, O. Brox, A. Grewe, M. Ibe, S. Jauns-Seyfried, C. Knieke, M. Körner, S. Küpper, M. Mauritz, H. Peters, A. Strasser, M. Vogel, and N. Weiss, "Managed and Continuous Evolution of Dependable Automotive Software Systems," in Proceedings of the 10th Symposium on Automotive Powertrain Control Systems, 2014, pp. 15–51.

[4] L. de Silva and D. Balasubramaniam, "Controlling Software Architecture Erosion: A Survey," Journal of Systems and Software, vol. 85, no. 1, Jan. 2012, pp. 132–151.

[5] B. Cool, C. Knieke, A. Rausch, M. Schindler, A. Strasser, M. Vogel, O. Brox, and S. Jauns-Seyfried, "From Product Architectures to a Managed Automotive Software Product Line Architecture," in Proceedings of the 31st Annual ACM Symposium on Applied Computing, ser. SAC'16. New York, NY, USA: ACM, 2016, pp. 1350–1353.

[6] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in 2007 Future of Software Engineering, ser. FOSE '07. IEEE Computer Society, 2007, pp. 55–71.

[7] R. Cloutier, G. Muller, D. Verma, R. Nilchiani, E. Hole, and M. Bone, "The Concept of Reference Architectures," Systems Engineering, vol. 13, no. 1, Feb. 2010, pp. 14–27.

[8] E. Y. Nakagawa, P. O. Antonino, and M. Becker, "Reference Architecture and Product Line Architecture: A Subtle but Critical Difference," in Proceedings of the 5th European Conference on Software Architecture, ser. ECSA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 207–211.

[9] E. Y. Nakagawa, F. Oquendo, and M. Becker, "RAModel: A Reference Model for Reference Architectures," in Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, ser. WICSA-ECSA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 297–301.

[10] E. Y. Nakagawa, M. Becker, and J. C. Maldonado, "Towards a Process to Design Product Line Architectures Based on Reference Architectures," in Proceedings of the 17th International Software Product Line Conference, ser. SPLC '13. New York, NY, USA: ACM, 2013, pp. 157–161.

[11] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo, "Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures," in Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture, ser. WICSA '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 143–152.

[12] J. van Gurp and J. Bosch, "Design Erosion: Problems & Causes," Journal of Systems and Software, vol. Volume 61, 2002, pp. 105–119.

[13] S. Herold and A. Rausch, "Complementing Model-Driven Development for the Detection of Software Architecture Erosion," in 5th Modelling in Software Engineering (MiSE 2013) Workshop at Intern. Conf. on Softw. Eng. (ICSE 2013), 2013.

[14] I. John and J. Dörr, "Elicitation of Requirements from User Documentation," in Ninth International Workshop on Requirements Engineering: Foundation for Software Quality. REFSQ '03, 2003.

[15] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, 2004.

[16] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns. Addison Wesley, 2001.

[17] J. Bosch, Design and use of software architectures: Adopting and evolving a product-line approach. Pearson Education, 2000.

[18] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," Journal of Systems and Software, vol. 74, no. 2, 2005, pp. 173–194.

[19] J. Bosch, "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization," in Proceedings of the Second International Conference on Software Product Lines, ser. SPLC 2. London, UK, UK: Springer-Verlag, 2002, pp. 257–271.

[20] A. Strasser, B. Cool, C. Gernert, C. Knieke, M. Körner, D. Niebuhr, H. Peters, A. Rausch, O. Brox, S. Jauns-Seyfried, H. Jelden, S. Klie, and M. Krämer, "Mastering Erosion of Software Architecture in Automotive Software Product Lines," in SOFSEM 2014: Theory and Practice of Computer Science, ser. LNCS, V. Geffert, B. Preneel, B. Rovan, J. Stuller, and A. M. Tjoa, Eds., vol. 8327. Springer, 2014, pp. 491–502.

[21] G. Holl, P. Grünbacher, and R. Rabiser, "A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines," Inf. Softw. Technol., vol. 54, no. 8, Aug. 2012, pp. 828–852.

[22] S. Thiel and A. Hein, "Modeling and Using Product Line Variability in Automotive Systems," IEEE Softw., vol. 19, no. 4, Jul. 2002, pp. 66–72.

[23] H. Holdschick, "Challenges in the Evolution of Model-based Software Product Lines in the Automotive Domain," in Proceedings of the 4th International Workshop on Feature-Oriented Software Development, ser. FOSD '12. ACM, 2012, pp. 70–73.

[24] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing PLA at Bosch Gasoline Systems: Experiences and Practices," in Software Product Lines. Springer, 2004, pp. 34–50.

[25] The Standish Group International, Inc., "CHAOS Chronicles 2003 report," West Yarmouth, MA, 2003.

[26] C. Knieke and M. Huhn, "Semantic Foundation and Validation of Live Activity Diagrams," Nordic Journal of Computing, vol. 15, no. 2, 2015, pp. 112–140.

[27] International Organization for Standardization, "ISO/DIS 26262: Road vehicles – functional safety," 2009.

[28] H. Peters, C. Knieke, O. Brox, S. Jauns-Seyfried, M. Krämer, and A. Schulze, "A Test-driven Approach for Model-based Development of Powertrain Functions," in Agile Processes in Software Engineering and Extreme Programming. 15th International Conference on Agile Software Development, XP 2014, G. Cantone and M. Marchesi, Eds. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 294–301.

[29] K. Beck, Test Driven Development. By Example. Addison-Wesley Longman, 2002.

[30] E. Lehmann, "Time Partition Testing – Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen," Ph.D. dissertation, Fakultät IV – Elektrotechnik und Informatik, TU Berlin, 2004.

[31] M. Mues, "Taint Analysis - Language Independent Security Analysis for Injection Attacks," Master's Thesis, TU Clausthal, Institute for Applied Software Systems Engineering, 2016.

[32] F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," Computer, vol. 20, no. 4, Apr. 1987, pp. 10–19.