Automotive Software Systems Evolution: Planning and Evolving Product Line Architectures

Axel Grewe, 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: {axel.grewe|christoph.knieke|marco.koerner|andreas.rausch|
mirco.schindler|arthur.strasser|m.vogel}@tu-clausthal.de

Abstract—**Automotive software systems are an essential and innovative part of nowadays connected and automated vehicles. The automotive industry is currently facing the challenge to re-invent the automobile. Consequently, automotive software systems, their software systems architecture, and the way we engineer those kinds of software systems are confronted with the challenge of managing complexity of the desired automotive software systems and the corresponding engineering process. We will present an approach that helps engineers to manage system complexity based on architecture design principles, techniques for architecture quality measurements and processes to iteratively evolve automotive software systems. Based on a running sample, we will demonstrate and illustrate the main assets of the proposed engineering approach.**

***Keywords*–Architecture Evolution; Software Product Lines; Software Erosion; Architecture Quality Measures; Automotive.**

I. Introduction

Usually many variants of a vehicle exist – different configurations of comfort functions, driver assistance systems, connected car services, or powertrains can be variably combined, creating an individual and unique product. To keep the vehicles cost efficient, modular components with a high reuse rate cross different types of vehicles are required. With respect to innovative and sophisticated functions, coming with the connected car and automated resp. autonomous driving the functional complexity, the technical complexity, and the networked-caused complexity is continuously and dramatically increasing. It is, and will be in future, a great challenge to further manage the resulting complexity.

Here, we propose an approach that helps engineers to manage functional software systems complexity based on modular, well-defined, and linked requirements as well as architectures. The goal is to create solid requirements and adequate architectures with the help of abstract principles, patterns, and describing techniques. In addition, we present a systematic approach for planning of development iterations and prototyping.

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 which is built according to the following:

1) Design principles for high cohesion
2) Design principles for abstraction and information hiding
3) Design principles for loose coupling

In the evolutionary development of automotive software systems, the range of functionalities grows steadily. Thus, the “essential” complexity of the architecture increases continuously due to the growth of the number of functions. However, the “accidental” complexity of the architecture of automotive software systems grows disproportionately to the essential complexity as illustrated in Figure 1 [1]. The growth of accidental complexity results from a “bad” architecture with strong coupling and a low cohesion which have evolved over the time. “Bad” architectures increase accidental complexity and costs, hinder reuseability and maintainability, and decrease performance and understandability. The three design principles for a good architecture mentioned above focus on the reduction of accidental complexity and on the changeability of the architecture.

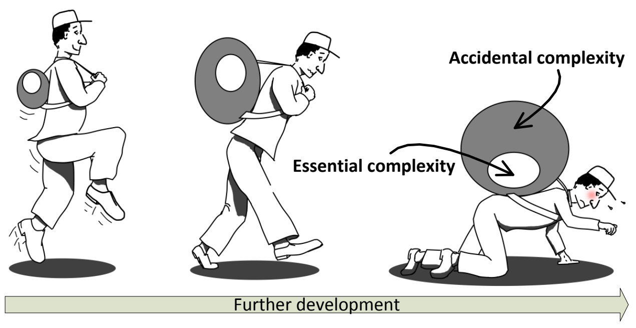

Figure 1. “Essential” vs. “Accidental” complexity

As an approach to manage software systems evolution, we propose three steps:

1) Methods and techniques for a good architecture design (Section IV-A)
2) Understanding of architecture and measuring of architecture quality (Section IV-B)
3) Systematic approach for planning of development iterations and prototyping (Section IV-C)

To make a statement about complexity, it is necessary to measure complexity and describe the results numerical. These numbers allow drawing conclusions from a system. Furthermore, it is necessary to describe complex relationships in a system. For this purpose, meaningful and understandable description techniques are needed. Such techniques allow complexity to be manageable. Finally, a systematic approach for planning of development iterations and prototyping is required.

The paper is structured as follows: Section III gives an overview on the related work. This paper refers to an overall development cycle for managed evolution of automotive software product line architectures that is proposed in Section II. In addition, Section II gives some formal definitions and introduces a real world example, a longitudinal dynamics torque coordination software, from automotive software engineering. Based on this example, we propose our methodology for planning and evolving automotive product line architectures in Section IV. Section V concludes.

II. Basics

A. Overall Development Cycle

Our methodology for managed evolution of automotive software product line architectures is depicted in Figure 2 (see [2]). The methodology consists of two development cycles which are executed concurrently: One cycle constitutes the development activities for product line development, whereas the second cycle is required for product specific development. Each cycle addresses the design of the logical architecture, the planning of development iterations and product releases, the implementation of software components, and architecture conformance checking.

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. A product on the other hand is a fully executable software status for a certain vehicle based on a project in conjunction with vehicle related parameter settings.

In the following 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 objectives of each of the 12 activities, including inputs and outputs:

Software system and software component requirements from requirements engineering (`PL-Requirements`) and artifacts of the developed product from the product cycle in Figure 2 (`P to PL`) serve as input to the management cycle of the product line architecture (PLA). Activities `PL-Design` and `PL-Plan` aim at designing, planning and evolving product line architectures and are explained in detail in this paper (see Section IV).

The planned implementation artifacts are implemented in `PL-Implement` on product line level whereas in `P-Implement` product specific implementation artifacts are implemented. For the building of a fully executable software status for a certain vehicle project, the project plan is transferred (`PL to P`) containing module descriptions and descriptions of the logical product architecture integration plan with associated module versions. In addition, special requirements for a specific project are regarded (`P-Requirements`). The creation of a new product starts with a basic planned product architecture commonly derived from the product line (`P-Design`). The product planning in `P-Plan` 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.

Each planned project 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 by architecture conformance checking for automotive software product line development. Furthermore, we apply architecture conformance checking to check conformance between the planned product architecture and the PLA in `P-Design`.

B. General structure and definitions

The relation between PLA, products, and modules is illustrated in Figure 3. We indicate the development points $t \in \mathbb{N}$ by the timeline at the bottom. Next, we give brief definitions of the terms PLA, product, and module.

PLA: On the top of Figure 3 the different versions of the PLA are illustrated. A PLA consists of logical architecture elements $l \in$ LAE (cf. `A`, `B`, `C` in Figure 3) and directed connections $c \in C$ between these elements. At each development point t exactly one version of the PLA exists. A certain PLA version is denoted by $\text{pla}_x \in PLA$, with $x \in \mathbb{N}$ to distinguish

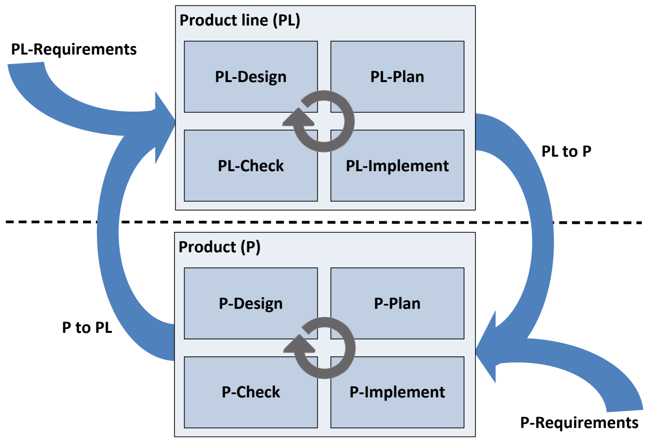

Figure 2. Overall development cycle

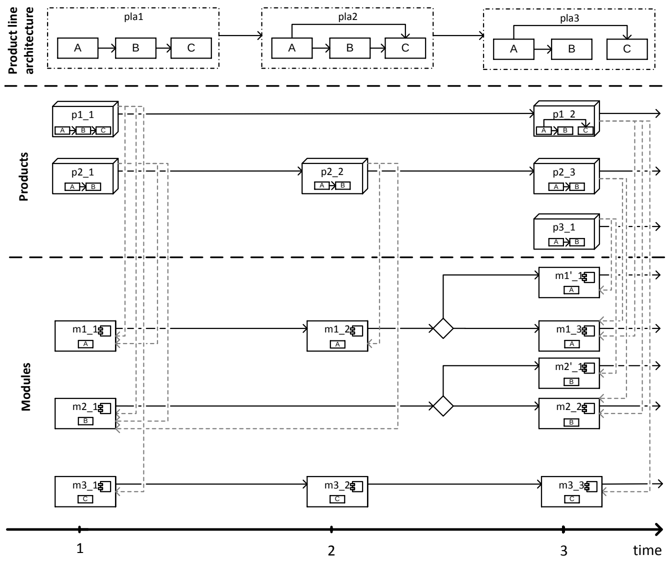

Figure 3. Relation between products, modules and PLA

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.

between PLA versions. The sequence of PLA versions is indicated by the arrows between the PLAs in Figure 3.

Product: A product $\mathrm{p}_{i\_j} \in P$ has a product identifier i and a version index j, with $i, j \in \mathbb{N}$. The sequence of versions is indicated by the flow relation between products in Figure 3. We assume a distinct mapping of p_{i_j} to a certain $\mathrm{pla}_x \in PLA$. A product p_{i_j} contains a product architecture $\mathrm{pa}_{i\_j} \in PA$, where pa_{i_j} is a subgraph of the corresponding pla_x. The set of corresponding modules of a product is indicated by the dashed arrows in Figure 3.

Module: A module $\mathrm{m}_{k\_l} \in M$ has a module identifier k and a version index l, with $k, l \in \mathbb{N}$. The sequence of versions is indicated by the flow relation between modules in Figure 3. We assume a distinct mapping of m_{k_l} to a certain $l \in \mathrm{LAE} \cup \{\perp\}$. By $\perp$ we allow m_{k_l} not to be assigned to a logical architecture element, called unbound m_{k_l}. A logical architecture element can be assigned to several modules, but a module can only be assigned to exactly one or no logical architecture element. A module $\mathrm{m}_{k\_l} \in M$ can belong to several products $\mathrm{p}_{i\_j} \in P$.

As illustrated in Figure 3, we assume a high degree of reuse: The same module version may be included in different products. Branches of the development path are depicted by the diamond symbol. Module $\mathrm{m}_{1'\_1}$ indicates a branch of the development path concerning module m_{1_3}.

C. Real World Example: Longitudinal Dynamics Torque Coordination

Our approach for designing the logical architecture described in the next section is based on our experience in the automotive environment. In numerous projects with the focus on software development for engine control units, we have developed architectural principles and concepts for architectural design and tested them on real sample projects. The following example shows frequent problems that arise as a result of strongly increasing accidental complexity. The approaches described in the next section are intended to help avoid the problems presented here by controlling the complexity. This paves the way for long-term maintenance and extensible architectures.

In our example, we consider the control of the braking and acceleration process, which is controlled by the driver via the brake and accelerator pedal, respectively. The implementation of these controls was originally carried out on completely separate developments. In the course of time, however, additional functions have been added: Not only the driver can act here by actuating the throttle or brake pedal. There are a number of additional functions, such as the ESP or ACC, which can act as accelerator and decelerator. In the case of longitudinal dynamics torque coordination (see Figure 4), both acceleration and braking processes must be coordinated with one another since there are interdependent interdependencies.

As a solution to the coordination problems, point-to-point connections between the software components were introduced, which however led to a strong increase in the accidental complexity: The realization of the reciprocal coordination

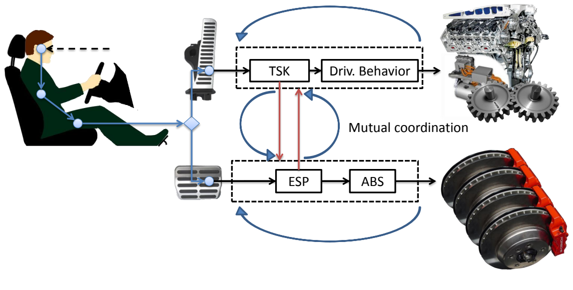

Figure 4. Automotive powertrain example: Mutual coordination

of the requesters was implemented in the example by the addition of a new explicit communication for the solution of coordination problems (see Figure 4, “mutual coordination”). In addition, existing functions had to be replicated in another context for the realization of the explicit communication. As a result, redundancies were created from the heads to the models. In addition, accidental complexity has increased disproportionately because of the wide interfaces and strong coupling within the architecture of the system.

III. Related Work

Next, we give an overview on the related work concerning software product line architecture design, evolution, and measurement of architecture quality. Mostly, we focus on approaches that are related to automotive and embedded software systems.

A. Software Product Line Architecture Design

In [3], reference architectures are assumed to be the basis for the instantiation of product line architectures (so-called family 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 [3] 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. Nakagawa et. al. define a reference model of reference architectures [4], and propose a methodology to design product line architectures based on reference architectures [5].

As discussed in [6] 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 [6]. In [7], 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 [8] proposes a method that brings together two aspects of software architecture: the design of software architecture and software product lines.

Thiel and Hein [9] 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.

Patterns and styles are an important means for software systems architecture specification and are widely covered in literature, see, e.g., [10][11]. However, architecture patterns are not explicitly applied for the development of automotive software systems yet. For automotive industry, we propose the use of architecture patterns as a crucial means to overcome the complexity.

B. Software Product Line Architecture Evolution

In order to enable the evolution of software product line architectures, architecture erosion has to be avoided. In [12], 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 [12] refers to architecture erosion for a single product architecture, whereas architecture erosion in software product lines is out of the scope of the paper.

Holdschick [13] 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 work in [14] proposes a systematic approach for managed and continuous evolution of dependable automotive software systems. They have identified three main challenges to strengthen automotive software systems engineering for the upcoming evolution: Complexity of automotive software systems and engineering processes has still to be manageable, flexibility has still to be provided, and dependability has still to be guaranteed. The work in [14] describes how complexity of automotive software systems can be managed by creating modular and stable architectures based on well-defined requirements.

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 [15][16]. In [15], 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 [16]. However, in both approaches a long-term minimization of erosion as well as a long-term evolution is not considered.

A previous version of our approach is described in [6] 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 in [6].

C. Measurement of Software Product Line Architecture Quality

To successfully plan and develop PLAs, it is necessary to measure key figures. These key figures are the basis for further developments. In [17], the SystEM-PLA framework is presented, which uses 98 metrics to assess the quality of a PLA. The analysis uses UML metrics to calculate key figures. A procedure is presented in [18] to measure NFA on PLAs. It is important to identify problems with regard to certain quality features as early as possible. The method uses different metrics to measure 3 NFAs: Maintainability, binary, and performance. The procedure results in possibilities to restrict products.

The work in [19] shows how traceability supports the evolution of SPL on feature level. For this purpose, a method is used to merge feature models, build files and source code with each other and to implement a change impact analysis by using metrics. As a result, erosion and problems are recognized at an early stage, and counter-measures can be taken.

In [20], PL are measured with the metric maintainability index (MI). The "Feature Oriented Programming" is used to map an SPL to a graph. The values are transformed into several matrixes. Next, singular value decomposition is applied to the matrixes. The metric maintainability index is then applied at different levels (product, feature, product line). The results show that by using the metric, features could be identified that had to be revised. The number of possible refactorings could be restricted.

In [21], several metrics are presented which are specifically used for measuring PLAs. The metrics are applied to "vADL" to determine the similarity, reusability, variability, and complexity of a PLA. The measured values can be used as a basis for further evolutionary steps.

IV. Planning and Evolving Automotive Product Line Architectures

A. *Concepts for Designing Automotive Product Line Architectures*

For the specification of software architectures design patterns, architectural patterns or styles are an important and suitable means, also in other engineering disciplines [10]. We subsume these under the term of architecture concepts. An architecture concept is defined as: "*a characterization and description of a common, abstract and realized implementation-, design-, or architecture solution within a given context represented by a set of examples and/or rules.*"

At the architectural level, these are often associated with terms as a client-server system, a pipes and filters design, or a layered architecture. An architectural style defines a vocabulary of components, connector types, and a set of constrains on how they can be combined [10]. To get a better understanding of the wide spectrum of architecture concepts typical samples of concepts are listed in the following:

- *Conventions:* naming, package/folder structure, vocabulary, domain model...
- *Design Patterns:* observer, factory, ...
- *Architectural Patterns:* client-server system, layered architecture, ...
- *Communication:* service-oriented, message based, bus, ...
- *Structures:* tiers, pipes, filters, ...
- *Security:* encryption, SSO, ...
- ...

Architectural concepts can be described in the form of classical patterns, by describing a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme specifies all constituent components, their responsibilities and relationships, and the way in which they will collaborate [11].

In the same way, we will illustrate some examples that we worked out in our automotive domain projects. Generally, the central issue is the increasing complexity of software systems with their technical and functional dependencies. A mapping of these dependencies to point-to-point connections will result in a huge, complex and difficult to maintain communication network. This leads to a likewise huge effort in the field of maintenance and further development for these software systems - small changes result in high costs.

This problem of a not manageable number of connections emerged in many industrial projects we explore for our field study. In the following we will present architectural concepts, which are addressing this problem in particular. Figures 5 and 6 show different components, whereby the components `Coordinator` and `Support` are atomic components and the components labeled as `Filter` are not atomic components, i.e., they can be decomposable.

1) Architecture Design Principle "Coordinator - PipesAndFilters - Support": The complexity of a component increases artificially with every new product, without integrating new functions. The reason for this phenomenon is due to the fact that each component had to calculate the system state for itself and this for each existing environment and product the component will be used in. In general, components are analyzing system data like sensor values for example and process them to realize their functionality. Thereby, it happens very often that a processing function is implemented several times. Besides data from other components are used, but this export data can change over time, so it can result in error states.

The design principle introduces a classification of data. If it is possible to classify the data, than it is possible to establish the typing of channels, as shown in Figure 5.

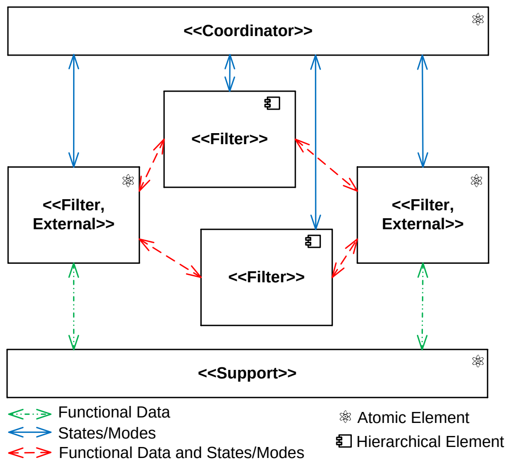

Figure 5. Architecture design principle: External elements

Each component has to declare a port for states and modes to uncouple the calculation of the system state from the component. A `Coordinator` component determines the global state for a set of components and uses the new defined port to coordinate the other components. The coordinator provides only states/modes and no functional data. A component in Figure 5 named as `Filter`, referring to the classical Pipes-and-Filters architecture pattern, can react to a state change automatically. Parameters are manipulated directly with the states/modes without an additional calculation. Components can be directly activated or stopped. The scheduling of the coordinator is independent from the scheduling of the other components, as each `Filter` checks the state/mode first. The functionality of the system is realized by the `Filter` components. For them it is allowed to exchange functional data as well as state and modes. Values which are required for the calculation within different components are provided by a so called `Support` component.

2) Architecture Design Principle “External Elements”: Today it is customary that not all components are developed in-house, some functions are implemented by external suppliers. But OEM components have requirements resulting in changes of interfaces, behavior or functionalities of theses external developed functions and components. It is not that easy to identify these external components on architectural level, but this information is essential for an economic development process because changes of external components are very effort and cost intensive.

Figure 5 shows a simple solution to handle external elements: `Filter` components which are developed external are annotated with `Filter, External`, so it is effortless to identify which component is external and which connections are affected.

3) Architecture Design Principle “Hierarchical Communication”: Over the time more and more components and functionality are added to a product. Different developers with different programming styles are working on the same product. Components without any reference to each other are organized in the same package or other organizational and structural units. Due to this accidental complexity it is not possible for a developer, system integrator or architect to get a well-founded knowledge of the whole system.

As presented in Figure 6, a `Filter` component can be decomposable, a so called non-atomic component contains a structure which follows the design principle visualized in Figure 5. It includes a `Coordinator` and `Support` component and an arbitrary number of `Filter` components. Whereby the inner `Filter` components have explicit defined responsibilities.

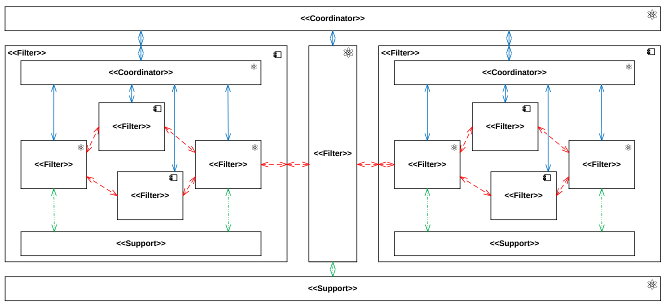

Figure 6. Architecture design principle: Hierarchical communication

By this design principle a repetitive structure on each abstraction level is established, which enables an easy and technical independent orientation in the whole system.

4) Architecture Design Principle “Component Model”: Components require knowledge about the behavior or the state/mode of the connected components. This results in a high coupling of components and the processing time increases, too.

As presented in Figure 7, a component consists of two parts with different responsibilities - `Execution control` and `Function algorithms`. Each part has a defined set of interfaces, types of communication channels, and exchange data.

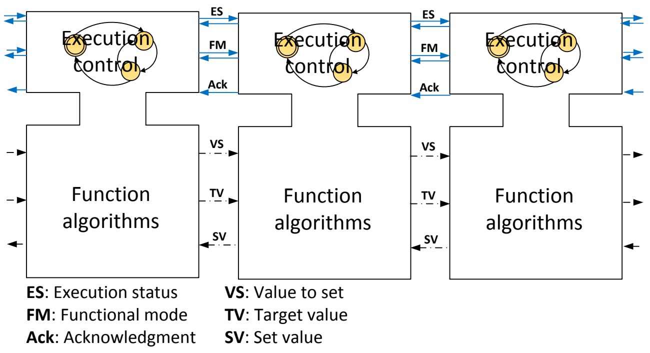

Figure 7. Communication scheme

The communication scheme is divided into two areas: the execution control and the functional algorithms. The execution control includes, on the one hand, the activation of the component, which is represented by the execution status. In addition, in the execution control, the functional mode (components internal mode) of the component is determined. The execution control sends an acknowledgment to the predecessor component when this component is active. The execution control communicates only by states/modes.

The function algorithms are processed when the execution status is set. Component specific values are calculated in the function algorithms. As output, they supply a manipulated variable and a target value. The manipulated variable is the value to set by the actuator. The target value is the value which is to be achieved in the future. The set value of the function algorithms is the value that is set by the controller. The functional algorithms only have functional data as input.

5) Architecture Design Principle “Feedback Channel”: The complexity of component-based control systems is increasing continuously, since there are more and more functional dependencies between the individual components. A mapping of these dependencies on point-to-point connections between the components results in a complex, hard-to-maintain communication network.

In component-based control engineering systems, control cascades are generated by connecting several components consecutively. The main data flow in this system is called the effect chain. In more complex systems, there are several effect chains that can partly overlap. In an effect chain, there are functional dependencies between components that are not directly connected one behind the other. To resolve these dependencies, additional point-to-point connections are added, which are technical dependencies between the components. The additional direct point-to-point connections between the components increase the coupling between the components and lead to a deterioration in the fulfillment of non-functional requirements, such as maintainability, understandability and extensibility. For example, the technical dependencies have to be taken into account in a further development. The worst case is a complete graph with cross-links between all components.

As a solution to this problem we introduce *feedback channels* (patent pending): The introduction of feedback channels enables the dissolution of functional dependencies without the introduction of technical point-to-point connections (see Figure 8). The feedback channel is parallel to the effect chain. Thereby, the necessary functional information is passed through the components of the effect chain. The feedback is directed against the effect direction. Components of an effect chain must provide feedback. This creates a technical communication network with which the functional information can be exchanged. Thus, there are only technical dependencies to neighboring components in the effect chain. The maintainability is improved as only technical dependencies on neighboring components in the effect chain have to be considered. Figure 8 shows the architecture design principle *feedback channel*.

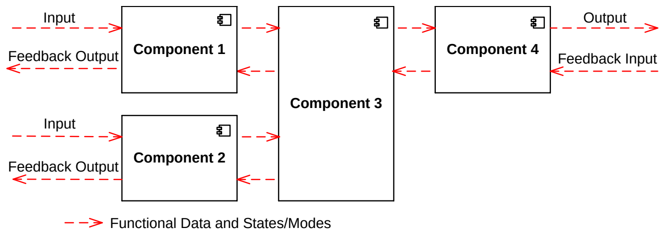

Figure 8. Architecture design principle: Feedback channel

All information / data from the end of the effect chain to the beginning of the effect chain are provided via the feedback. Thus, a component can adapt itself to the current situation in the effect chain without the necessity to create an explicit connection to all components in the effect chain. Furthermore, only the dependency of a component to the adjacent components of an effect chain exists. If the processing order of the components is selected s.t. all inputs are processed first and then the feedback, all components of the effect chain have the information on the current system state available in the next computing cycle. The effect chain to Figure 8 then looks as follows: The four components process their inputs in the effect direction. The components are then processed in the reverse order and the feedback is processed, i.e., from `Component 4` to `Component 1`. Here, components 1 and 2 can be interchanged in their processing.

In summary, the overall system is more maintainable and easier to expand by this architecture design principle. The individual components do not have to be connected to all components in order to know the system state. Through the feedback channel there is an information exchange between all components in the same computing cycle. Controllers can adapt themselves directly if they do not have access to an actuator.

Summary

The presented architectural concepts in this section were developed within different industrial projects in the automotive domain involving different software architects and project members. Nevertheless, there are similarities between the presented concepts, which become explicit by generalization and the representation by a uniform description language. Thereby, the projects focused the same as well as varying problem issues and requirements. With this representation technique it was possible to reuse the concepts in other projects to increase the quality in an early phase of development and to economize effort, because the projects start discussing about architectural concepts.

The architectural concepts presented in this paper are developed iteratively and in some cases the development time took over one year. As a result from our field study we can outline that there are similarities between the architectural evolution of product lines and the abstract and generic development process of concepts which is not surprising. The evolution of an architectural concept looks like the same - reuse and adaptation in other projects, which sometimes results in a new concept. Besides we can observe that the different levels of abstraction we have for architecture descriptions, we can find for concepts, as well. For example, the architecture design principle IV-A4 (Component Model, Figure 7), describes coordinating functionality, status and mode information and functional data connections. All these aspects we can find in the design principle IV-A2 (Coordinator, PipesAndFilters, Support, Figure 5), too. With the difference that the `Component Model` concept is for low level control functions, whereas the other concept deals with components on another abstraction level - to clarify the `Component Model` principle can be applied for a `Filter`, for example.

Architecture concepts like the ones presented before and all other aspects mentioned in the introduction of this section, especially the specification of wording and naming conventions help to build a collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practice [11].

The result of making these concepts explicit on this abstraction level leads to discussions about architectural problems and generic solution schemes. In particular at the product line architecture level the focus is shifted from the more technical driven problems upon the more abstract and software architecture oriented issues. Over time this leads to new architectural concepts, which are documented, evaluated, maybe extracted from existing products, but making them explicit and integrating them at the right places in the further development process.

Another very important aspect dealing with architectural concepts is the monitoring of the concrete realizations of them. In our approach the `Check` activity takes care of it. All the presented concepts can be represented by a logical rule set, as described in [22]. Related to the fact that all elements of the software are subjects to the evolution process, architectural concepts can change or had to be adapted over time. This means that the violation of an architectural rule indicates not always a bad or defective implementation, it can additionally give the impulse to review the associated concept and the context. In our approach the assessment of the rule violation is included in the `Check` activity and if there is an indication for a rule adaptation this will be analyzed and worked out in detail in the next `Design` activity. Overall it leads to a managed evolution.

B. Understanding of Architecture and Measuring of Architecture Quality

Software development is an evolutionary and not a linear process. The costs caused by errors in software in the last years, especially in the automotive industry, are very high (15-20% form earnings before interest and taxes). Thus, it is necessary to understand and evaluate the architecture to support further development. In a vehicle, software will occupy a larger and larger part and the costs caused by errors will be rising. Therefore, it is important to control the quality of the software continuously. Problems/Errors can be detected early so that the quality of the software increases. The quality of the software depends in particular on the quality of the corresponding software architecture. In our approach, we use PLAs for automotive software product line development. PLAs are special types of software architectures. They do not only describe one system, but many products which can be derived from this architecture. Variability of the architecture, reuse of products, and the complexity are important values to assess the quality of this architecture.

Today, metrics mainly focus on code level. The most common metrics are *Lines of Code*, *Halstead*, and *McCabe*. In object-oriented programming (OOP), *MOOD metrics* and *CK metrics* are used. However, these metrics are not suitable for measuring PLAs. For assessing a PLA, the most important value is variability, as the degree of variability increases complexity in PLAs. Further important values are complexity and maintainability of the possible products and the PLA. As products shall be reused for other products, a high reuse-rate of products is an important objective of the PLA. A high reuse-rate also implies a high focus on maintainability of the products.

In our approach, we assess the *modification effort*, *reuse rate* and *cohesion* of a PLA, since we can thus evaluate the properties discussed above. In the following, we give formulas for the calculation of modification effort, reuse rate and cohesion. Here, we refer to the definitions of Section II-B, and the system structure depicted in Figure 3.

1) Modification effort: The modification effort measures the effort caused by the planned changes in the PLA: How many logical architecture elements (LAE), and products are affected by the change? The calculated result value is between 0 (no elements have to be changed) and 1 (all elements have to be changed). Simple changes can have a high impact to products and modules. The value supports the architect to improve understanding the architecture. Maybe there is a better solution to design the new PLA with less modification effort. The modification effort $\mathcal{E}$ to develop a new PLA version pla_{x+1} for a given PLA pla_x is calculated as follows on the level of PLA and products:

$$\mathcal{E}^{PLA} = \frac{number\ of\ concerned\ LAE}{number\ of\ all\ LAE} \tag{1}$$

$$\mathcal{E}^{P} = \frac{number\ of\ concerned\ products}{number\ of\ all\ products} \tag{2}$$

where $concerned\ LAE/products$ denote the logical architecture elements/products that have to be modified or added/deleted when introducing the new PLA version. In Table II we apply $\mathcal{E}$ on the example in Figure 3.

TABLE II. MODIFICATION EFFORT FOR THE EXAMPLE OF FIGURE 3.

$\mathcal{E}$	$pla_1 \rightarrow pla_2$	$pla_2 \rightarrow pla_3$
$\mathcal{E}^{PLA}$	$\frac{\lvert\{A,C\}\rvert}{\lvert\{A,B,C\}\rvert} = \frac{2}{3}$	$\frac{\lvert\{B,C\}\rvert}{\lvert\{A,B,C\}\rvert} = \frac{2}{3}$
$\mathcal{E}^{P}$	$\frac{\lvert\{p_1,p_2\}\rvert}{\lvert\{p_1,p_2\}\rvert} = \frac{2}{2} = 1$	$\frac{\lvert\{p_1,p_2,p_3\}\rvert}{\lvert\{p_1,p_2,p_3\}\rvert} = \frac{3}{3} = 1$

Consider, e.g., step $pla_1 \rightarrow pla_2$ in Table II: Note that each module is assigned to only one LAE in this example. Hence, modules are not considered in this example. In practice an LAE can be assigned to several modules to realize functionality. In this step the architect adds a connection between the LAE A and LAE C on the PLA. The modification effort for the PLA is $\frac{2}{3}$, because two of three LAE are affected by this change. On product level the modification effort $\mathcal{E}^P$ is 1: p_{1_1} and p_{2_1} contain LAE A and are thus affected. Note that for $\mathcal{E}^P$ we do not specify the version index in the calculation in Table II.

In this example, all products are affected by the modification in both development steps. There is no other way to reduce the modification effort. However, new product versions are not released at each point in time even if the product is concerned by the PLA modification (see product p_1 at $time = 2$ in Figure 3).

2) Reuse rate: To keep the vehicles cost efficient, modular products with a high reuse rate cross different types of vehicles are desired. The aim is to reuse modules in different products. The reuse rate $\mathcal{R}^m$ of a module m in a certain PLA version pla_x is calculated as follows:

$$\mathcal{R}^m = \frac{number\ of\ usage\ of\ m\ in\ all\ products\ of\ pla_x}{number\ of\ all\ products\ of\ pla_x} \tag{3}$$

Average reuse rate $\mathcal{R}^M$:

$$\mathcal{R}^M = \frac{\sum \mathcal{R}^m}{number\ of\ all\ modules} \tag{4}$$

In Table III we apply $\mathcal{R}$ on the example in Figure 3.

TABLE III. REUSE RATE FOR THE EXAMPLE OF FIGURE 3.

$\mathcal{R}$	pla_1	pla_2	pla_3
$\mathcal{R}^{m_1}$	$\frac{2}{2}$	$\frac{1}{1}$	$\frac{2}{3}$
$\mathcal{R}^{m_2}$	$\frac{2}{2}$	$\frac{1}{1}$	$\frac{2}{3}$
$\mathcal{R}^{m_3}$	$\frac{1}{2}$	$\frac{0}{1}$	$\frac{1}{3}$
$\mathcal{R}^{m'_1}$	–	–	$\frac{1}{3}$
$\mathcal{R}^{m'_2}$	–	–	$\frac{1}{3}$
$\mathcal{R}^{M}$	$\frac{5}{2}/3 \approx 0.84$	$\frac{2}{1}/3 \approx 0.67$	$\frac{7}{3}/5 \approx 0.47$

Consider, e.g., pla_1 and $\mathcal{R}^{m1}$ in Table III: Modules m_{1_1} and m_{2_1} are both used in products p_{1_1} and p_{2_1}. Thus, the reuse rate is $\frac{2}{2} = 1$ (100%). In the example the average reuse rate for pla_1 is 0.84 (84%). This value constitutes a high degree of reuse. For pla_3 and $\mathcal{R}^{m1}$ the reuse rate has to take the new product p_{3_1} into account. As m_{1_3} is used in two products and the number of products is three, $\mathcal{R}^{m1} = \frac{2}{3}$ ($\approx$ 67%).

In the example the average reuse rate in pla_3 is 0.47. The comparison between pla_1 and pla_3 shows that the reuse rate has deteriorated. This is to be expected since new products and modules are added. In the next planning phase of a new PLA these new modules should be used in more products to increase the reuse rate.

3) Cohesion: A high cohesion is preferable. The value for cohesion denotes the rate, how many export values of the modules are used inside a product. The higher the value, the better the cohesion of the product. We call export and import values of modules *exports* and *imports* in the following.

The cohesion $\mathcal{A}^p$ of a product p is calculated as follows:

$$\mathcal{A}^p = \frac{number\ of\ all\ exports\ of\ all\ modules\ used\ in\ p}{number\ of\ all\ exports\ of\ all\ modules\ in\ p} \quad (5)$$

The average cohesion $\mathcal{A}^P$ of products of a PLA version is calculated as follows:

$$\mathcal{A}^P = \frac{\sum \mathcal{A}^p}{number\ of\ all\ products} \quad (6)$$

The cohesion of the PLA $\mathcal{A}^{PLA}$ is calculated as follows:

$$\mathcal{A}^{PLA} = \frac{number\ of\ all\ exports\ of\ modules\ used\ in\ all\ products}{number\ of\ all\ exports\ of\ all\ modules\ of\ all\ products} \quad (7)$$

In the following Table IV, we set randomly chosen values for exports and imports at $time = 1$ for the modules. We assume that the architect has access to the whole information of LAE, all products, and all modules at this time.

TABLE IV. EXPORTS AND IMPORTS AT TIME=1 IN FIGURE 3.

Module	*Number of export values*	*Number of import values*
m_{1_1}	3	1
m_{2_1}	4	3
m_{3_1}	2	3

TABLE V. COHESION FOR THE EXAMPLE OF FIGURE 3.

$\mathcal{A}$	pla_1	pla_2	pla_3
$\mathcal{A}^{p_1}$	$\frac{1+1+0}{3+4+2} \approx 0.22$	–	$\frac{2+0+0}{3+4+2} \approx 0.22$
$\mathcal{A}^{p_2}$	$\frac{1+0}{3+4} \approx 0.14$	$\frac{1+0}{3+4} \approx 0.14$	$\frac{1+0}{3+4} \approx 0.14$
$\mathcal{A}^{p_3}$	–	–	$\frac{1+0}{3+4} \approx 0.14$
$\mathcal{A}^P$	≈ 0.18	≈ 0.14	≈ 0.17
$\mathcal{A}^{PLA}$	$\frac{1+1+0+1+0}{3+4+2+3+4} \approx 0.19$	$\frac{1+0}{3+4} \approx 0.14$	$\frac{2+0+0+1+0+1+0}{3+4+2+3+4+3+4} \approx 0.17$

Consider, e.g., pla_1 and $\mathcal{A}^{p_1}$ in Table V: Product p_{1_1} has three modules (m_{1_1}, m_{2_1}, m_{3_1}). In product p_{1_1} LAE A has a connection (export) to B and B has a connection (export) to C. In Table IV all export values are listed. The cohesion is calculated as follows:

$$\frac{\sum used\ exports\ of\ m_{1\_1}, m_{2\_1}, m_{3\_1}}{\sum all\ exports\ of\ m1\_1, m2\_1, m3\_1} = \frac{1+1+0}{3+4+2} \approx 0.22$$

For a whole PLA all used export values of modules in all products are aggregated. The result for pla_2 shows that the change operation concerns all products and a part of the LAE and modules. The expected cohesion in pla_3 is worse compared to pla_1. The quality of the PLA has slightly deteriorated. Modules realize more than one functionality because they are used in more than one project. Therefore, cohesion is competing to the reuse rate. It is planned to evaluate these metrics and determine the intervals of the values for "good" and "bad" with the help of experts in one of our industrial projects.

4) Applying change operations on a PLA: A software architect changes the PLA to fulfill new requirements. The aim is to implement the new requirements with the least possible adaptation on the product/module level.

Figure 9 exemplarily describes the procedure of applying change operations on a PLA. The procedure starts with the current PLA and all products and modules at $time = 1$. To make change operations, the software architect performs the following steps:

1) The architect adds a new change operation to the PLA.
2) The above metrics are performed on the intermediate PLA b. The results are considered as bad by the architect and the changes are rejected.
3) The architect adds a new change operation to the PLA. The above metrics are performed on the intermediate PLA. The results are evaluated as good and the PLA c serves as the basis for the next step.
4) The architect adds a new change operation to the PLA c.
5) The above metrics are performed on the intermediate PLA d. The results are considered as bad by the architect and the changes are rejected.
6) The architect adds a new change operation on the PLA c resulting in PLA e. Again, the metrics are applied. The results are rated as good. As all requirements have been implemented, PLA e is the new PLA vision and serves as input for the planning.

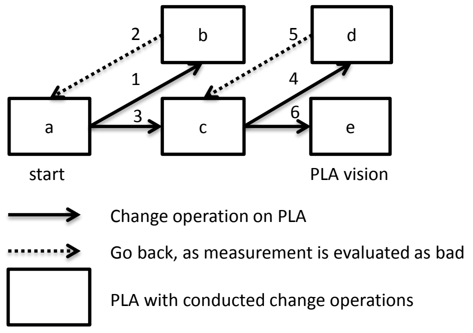
2
b
5
d
1
4
a
3
c
6
e
start
PLA vision
Change operation on PLA
Go back, as measurement is evaluated as bad
PLA with conducted change operations

Figure 9. Example: Applying change operations on a PLA

C. Planning of Development Iterations and Prototyping

In our case the planning of the further development involves several activities, e.g., performing planning of each modification of PLA and PA. The problem arises when `PL-Requirements` or `P-Requirements` needs to be realized within certain development time and within certain development costs. Planning solves the problem by defining timed activities considering the effort limitations.

Planning consists of a sequence of iterations. Iterations are defined as a number of architecture elements that must be realized in a time period bounded by t_{start} and t_{end} with $t_{start}, t_{end} \in \mathbb{N}, t_{start} < t_{end}$. Within each time period the activities `Design`, `Plan`, `Implement` and `Check` are ordered. The iteration is completed when all modifications are realized by `Design`, `Implement`, and checked to be conform to architecture rules by `Check`. An example of a sequence of three iterations is shown in Figure 3. In Figure 3, the expected result of modifications on PLA at several time points is defined, which corresponds to `PL-Plan`. Moreover, the expected result of modifications on PA are defined where products, modules and their mapping for three time points is shown in Figure 3.

The effort caused to realize the planned number of architecture elements is estimated by the activities `Design` and `Implement`, to achieve the PLA and PA development within given effort limitations. In case of a deviation between planned and actual estimations the initial plan is modified. Therefore, effort estimations are made by considering the necessary effort of PLA or PA modifications from `Design` and from `Implement`. In the following, details about effort estimations according to PLA and PA modifications are presented to achieve estimation based planning.

The first estimation concept is based on metrics to evaluate the modification effort. For example, modification effort according to connection structure and component structure is estimated by rating cohesion of components. Another estimation concept is to evaluate the effort based on modification realizing a new pattern in the appropriate PLA or PA. Hence, simple connection or component related modifications are lightweight, pattern based structure modifications are heavyweight. Modifications rated as heavyweight often involve a huge number of architecture components and products. Therefore, in such a case our methodology suggests to outsource such heavyweight modifications into a prototype projects. This special case is enclosed by the activity `PL to P` of our methodology.

V. Conclusion

We introduced a sophisticated approach for automotive software systems evolution by concepts for planning and evolving product line architectures. To manage functional software systems complexity we proposed an approach based on modular, well-defined, and linked requirements as well as architectures. First, we proposed methods and concepts to create adequate architectures with the help of abstract principles, patterns, and describing techniques. Such techniques allow making complexity manageable. Next, we suggested techniques for understanding of architecture and measuring of architecture quality. With the help of numerical results of these measurements, we can make a statement about complexity, as well as conclusions about a system. Finally, we described how to plan development iterations and prototyping. We demonstrated our concepts by examples especially from the automotive domain.

References

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

[2] C. Knieke et al., “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, 2016, accepted.

[3] R. Cloutier et al., “The Concept of Reference Architectures,” Systems Engineering, vol. 13, no. 1, Feb. 2010, pp. 14–27.

[4] E. Y. Nakagawa, F. Oquendo, and M. Becker, “RAModel: A Reference Model for Reference Architectures,” in Proc. of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, ser. WICSA-ECSA ’12. IEEE Computer Society, 2012, pp. 297–301.

[5] 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. ACM, 2013, pp. 157–161.

[6] B. Cool et al., “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.

[7] A. Strasser et al., “Mastering Erosion of Software Architecture in Automotive Software Product Lines,” in SOFSEM 2014: Theory and Practice of Comp. Sc., ser. LNCS, vol. 8327. Springer, 2014, pp. 491–502.

[8] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000.

[9] 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.

[10] M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc., 1996.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing, 1996.

[12] 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.

[13] 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.

[14] A. Rausch et al., “Managed and Continuous Evolution of Dependable Automotive Software Systems,” in Proceedings of the 10th Symposium on Automotive Powertrain Control Systems, 2014, pp. 15–51.

[15] B. Hardung, T. Kölzow, and A. Krüger, “Reuse of Software in Distributed Embedded Automotive Systems,” in Proc. of the 4th ACM intern. conf. on Embedded software. ACM, 2004, pp. 203–210.

[16] M. Steger et al., “Introducing PLA at Bosch Gasoline Systems: Experiences and Practices,” in Software Product Lines. Springer, 2004, pp. 34–50.

[17] A. G. Chiquitto, I. M. S. Gimenes, and E. Oliveira, “Symples-cvl: A sysml and cvl based approach for product-line development of embedded systems,” in Proceedings of the 2015 IX Brazilian Symposium on Components, Architectures and Reuse Software, ser. SBCARS ’15. IEEE Computer Society, 2015, pp. 21–30.

[18] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake, “Measuring non-functional properties in software product line for product derivation,” in Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference, ser. APSEC ’08. IEEE Computer Society, 2008, pp. 187–194.

[19] L. Passos et al., “Feature-oriented software evolution,” in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS ’13. ACM, 2013, pp. 17:1–17:8.

[20] G. Aldekoa, S. Trujillo, G. S. Mendieta, and O. Díaz, “Quantifying Maintainability in Feature Oriented Product Lines,” in Proceedings of the 12th European Conference on Software Maintenance and Reengineering. IEEE, 2008, pp. 243–247.

[21] T. Zhang, L. Deng, J. Wu, Q. Zhou, and C. Ma, “Some Metrics for Accessing Quality of Product Line Architecture,” in 2008 International Conference on Computer Science and Software Engineering, vol. 2, 2008, pp. 500–503.

[22] S. Herold, “Architectural Compliance in Component-Based Systems. Foundations, Specification, and Checking of Architectural Rules.” Ph.D. dissertation, Technische Universität Clausthal, 2011.