Automotive Software Product Line Architecture Evolution: Extracting, Designing and Managing Architectural Concepts

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*—The amount of software in cars has been growing exponentially since the early 1970s, and one can expect this trend to continue. To keep the software development for vehicles cost efficient, modular components with a high reuse rate cross different types of vehicles are used. Often, a product line approach is used to handle variability. As the underlying software product line architecture and its evolution are generally not explicitly documented and controlled, architecture erosion and complexity within the software product line architecture are growing steadily. In the long-term, this leads to reduced reusability and extensibility of the software artifacts, and thus, to a deterioration of evolvability. First, we propose methods used to extract initial product line architectures by recovery/discovery methods and describe our experiences gained from a real world example. Furthermore, we integrate this approach into an evolutionary incremental development process and show how a knowledge based process for architecture evolution and maintenance for architectural concepts can be implemented. The approach includes methods and concepts to create adequate architectures with the help of abstract design principles, patterns, and description techniques. Our approach helps software engineers to manage system complexity by suitable architectural concepts, by techniques for architecture quality measurements and by processes to iteratively evolve automotive software systems. We demonstrate our approach on a real world example, the longitudinal dynamics torque coordination from automotive software engineering.**

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

I. Introduction

This paper is a substantial extension of the work presented at the ADAPTIVE 2017 conference [1]. 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.

As the number of functions grows steadily in the evolutionary development of automotive software systems, the “essential” complexity of the product line architecture increases continuously. However, the “accidental” complexity of the architecture of automotive software systems grows disproportionately to the essential complexity as illustrated in Figure 1 [2]. The growth of accidental complexity results from a “bad” architecture (product line architecture and product architecture) with strong coupling and a low cohesion, which have evolved over the time. “Bad” architectures increase accidental complexity and costs, hinder reusability and maintainability, and decrease performance and understandability.

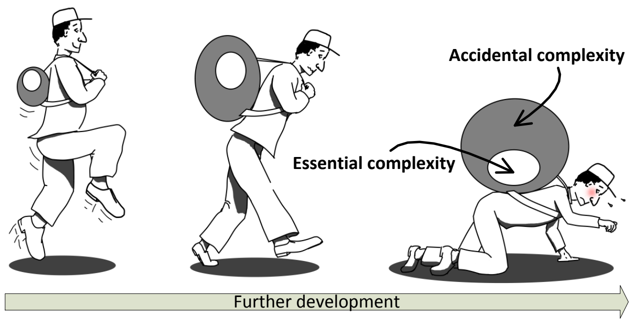

Figure 1. “Essential” vs. “Accidental” complexity

A software system architecture defines the basic organization of a system by structuring different architectural elements and relationships between them. The reduction of accidental complexity of the software system architecture is crucial for the success of the system to be developed and its evolvability. By our definition, a “good” architecture is a modular and evolvable architecture, which should be built according to the following design principles:

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

In this paper, we propose a sophisticated approach for extracting, designing and managing architectural concepts and thus enabling long-term evolution of automotive software product line architectures. By the term “architectural concepts” we subsume design patterns, architectural patterns or styles (see Section V). Our approach helps engineers to manage functional software systems complexity based on adequate architectures with the help of abstract principles, patterns, and description techniques. As an approach to manage automotive software product line architecture evolution, we propose the following steps:

(1) We often have to deal with an initially eroded software architecture which first has to be repaired. Thus, we propose methods used to extract initial product line architectures by recovery/discovery methods and describe our experiences gained from a real world example. The recovery/discovery approach is supported by an approach to extract architectural concepts from system realizations. (Sections IV and V)

(2) For designing automotive software product line architectures, we present architectural concepts developed within different industrial projects in the automotive domain involving different software architects and project members. Here, we aim to build the architecture according to the three design principles for a "good" architecture design mentioned above. In addition, we propose metrics to measure complexity of the design. Finally, a systematic approach for planning of development iterations and prototyping is introduced. (Section VI)

(3) Furthermore, we integrate this approach into an evolutionary incremental development process and show how a knowledge-based process for architecture evolution and maintenance for architectural concepts can be implemented. The term "knowledge-based" in this context means, that knowledge-based techniques like knowledge management are applied in the process. (Section VIII)

Step (1) of our approach is optional and only required in the case of an eroded software architecture, i.e., it is not intended for software product lines that are newly developed.

The paper is structured as follows: Section II gives an overview on the related work. Our overall development cycle for managed evolution of automotive software product line architectures is proposed in Section III. The first process activity to extract the initial architecture is proposed in Section IV. Section V introduces a new approach to extract concepts from source models. In Section VI we propose our methodology for designing and planning automotive product line architectures including long-term evolution. Section VII introduces a real world example, a longitudinal dynamics torque coordination software, from automotive software engineering. We apply our methodology for planning and evolving automotive product line architectures on this example and present the results of a corresponding case study. Section VIII extends the proposed methodology by an approach for knowledge-based architecture evolution and maintenance. Section IX concludes.

II. Overview on the Related Work

To the best of our knowledge no continuous overall development cycle for automotive software product line architectures exists. Next, we give an overview on the related work. Mostly, we focus on approaches that are related to automotive and embedded software systems.

A. Software Erosion

Van Gurp and Bosch [3] 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.

In [4], 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. Those rules can, e.g., contain structural information about the software like allowed communications. In [4], the rules are expressed as logical formulas, which can be evaluated automatically to the compliance to the product line architecture (PLA). These rules are extracted via Architecture Checker (ArCh) framework [5].

In order to enable the evolution of software product line architectures, architecture erosion has to be avoided. In [6], 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. The approaches discussed in [6] are primarily classified into three generic categories that attempt to minimize, prevent and repair architecture erosion. The categories are refined by a set of strategies to tackle erosion: process-oriented architecture conformance, architecture evolution management, architecture design enforcement, architecture to implementation linkage, self-adaptation and architecture restoration techniques consisting of recovery, discovery and reconciliation. However, each approach discussed in [6] refers to architecture erosion for a single product architecture, whereas architecture erosion in software product lines is out of the scope of the paper. Furthermore, as discussed in [6], none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

B. Software Product Line Architecture Extraction

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 [7] 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 [8]. In [9] 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.

C. Software Product Line Architecture Evolution and Life-Cycle Management

The work in [10] elaborates on the foundations of software product line engineering and provides experience-based knowledge about the two key processes, domain engineering and application engineering, and the definition and management of variability.

Holdschick [11] 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.

D. Reference Architectures

In [12], reference architectures are assumed to be the basis for the instantiation of PLAs (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 [12] 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. discuss the differences between reference architectures and PLAs 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 [13]. Furthermore, they define a reference model of reference architectures [14], and propose a methodology to design PLAs based on reference architectures [15], [16].

E. Software Product Line Architecture Design

Patterns and styles are an important means for software systems architecture specification and are widely covered in literature, see, e.g., [17], [18]. 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.

The work in [19] proposes a method that brings together two aspects of software architecture: the design of software architecture and software product lines. Deelstra et al. [20] 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.

Thiel and Hein [21] 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.

Flores et. al. [22] explain the application of 2GPLE (Second Generation Product Line Engineering) - an advanced set of explicitly defined product line engineering solutions - at General Motors.

F. Measurement of Software Product Line Architecture Quality

Siegmund et al. [23] present an approach for measuring non-functional properties in software product lines. The results are used to compute optimized software product line (SPL) configurations according to user-defined non-functional requirements. The method uses different metrics to measure three non-functional properties: *Maintainability*, *Binary Size*, and *Performance*. Siegmund et al. also discuss and classify the presented techniques to measure non-functional properties of software modules.

Passos et al. [24] show how automatic traceability, analyses, and recommendations support the evolution of SPL in a feature-oriented manner. They propose among other things a change-impact analysis to assess or estimate the impact and effort of a change. Furthermore, they regard metrics for architectural analysis. As a result, erosion and problems can be recognized at an early stage, and counter-measures can be taken. The ideas are illustrated by an automotive example.

In [25], product lines 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 matrices. Next, singular value decomposition is applied to the matrices. The metric MI 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 [26], several metrics are presented, which are specifically used for measuring PLAs. The metrics are applied to “vADL”, a product line architecture description language, to determine the similarity, reusability, variability, and complexity of a PLA. The measured values can be used as a basis for further evolutionary steps.

G. Approaches on Multi Product Lines

The work in [27] 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.

H. Reuse of Software Artifacts in the Automotive Domain

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 [28], [29]. In [28], 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 [29]. However, in both approaches a long-term minimization of erosion as well as a long-term evolution is not considered.

III. Basics

In this section we introduce our overall development cycle for managed evolution of automotive software product line architectures.

A. 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 for repairing an eroded software (see Section IV). 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

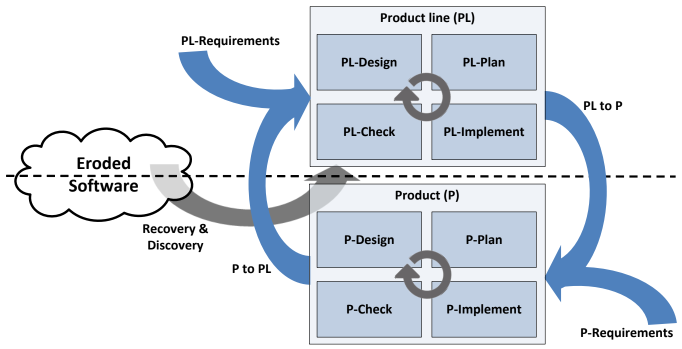

Figure 2. Overall development cycle

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 product line architecture is developed in parallel (activity `PL-Design`). The large arrows between the two development levels indicate transitions requiring an external decision-making process: The decision to start a new product development or prototyping (activity `PL to P`), and the inclusion and generalization of lessons learned during product development in the evolution of the SPL (activity `P to PL`), respectively.

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 runnable software status for a certain vehicle that can be flashed and executed on an ECU and is 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 13 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 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 VI).

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 $\mathrm{pla}_x \in PLA$, with $x \in \mathbb{N}$ to distinguish 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.

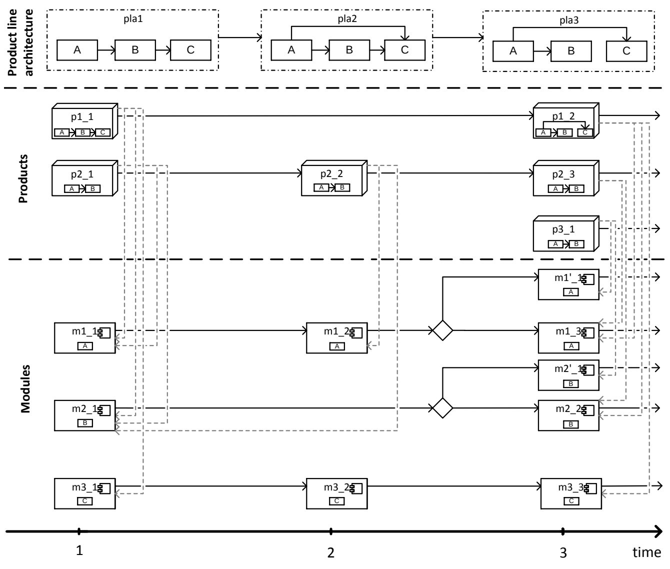

Figure 3. Relation between products, modules and PLA

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.
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.

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}.

IV. MAKING THE ARCHITECTURE EXPLICIT

With a high degree of erosion, a further development of the software is only possible at great effort. Before approaches to minimize erosion can be applied, the architecture must first be repaired. In this section, we investigate how approaches for architecture extraction can be adapted to be applied to automotive software product line architectures. First, we propose methods used to extract initial architectures. Next, in the second subsection, we give results and our experiences gained from a real world example.

A. Methods Used to Extract Initial Architectures and their Application

In this section we propose an approach for repairing an eroded software consisting of a set of product architectures (PAs) by applying strategies for recovery and discovery of the PLA (see left part of Figure 2). *Recovery* uses reverse engineering techniques to extract the implemented architecture from source artifacts, and *discovery* hypothesizes its intended architecture [6]. The proposed approach constitutes a substantial extension of the work presented in [30], where only a brief idea of the approach is introduced without any experimental results.

An explicit PLA definition constituting the top level architecture is important to coordinate the shared development between the OEM and the suppliers. Each product that is developed has a PA whose structure should be mapped onto the top level architecture. This top level architecture describes the structure of all realizable PAs. However, because of software sharing an overall assignment of top level groups to modules, and their interface, is missing. The knowledge of the overall, product independent structure is not explicitly documented, and therefore exists only implicitly in the minds of the participants. Further development of existing products and the development of new products lead to eroded PAs as an initially demanded structure is not available.

As a major challenge, we have to deal with product line development where a set of software components - so called *modules* - constitutes the basis for deriving a huge number of products. Therefore it is necessary to know about the derivable PAs from a given PLA. Two PLAs are distinguished: Current derivable PAs are captured by the *actual PLA* (APLA). All planned PAs for future development are captured by the *target PLA* (TPLA). In the Recovery & Discovery activity we recover the APLA and discover a TPLA candidate.

In the *Recovery & Discovery* activity we are using domain specific expertise and architecture related data from a repository to create the two PLAs. Figure 4 shows how the TPLA (*step d)*) and APLA (*step e)*) are created. For this purpose the APLA relevant elements are described by the recovered structure from data mining (*step b)*) and from functional analysis (*step c)*) using a set of PAs. The PAs are provided by *step a)*. Due to the ease of handling in the first iteration of *step a)* only some products are selected from the data dictionary. The following iterations extend the scope to more products. In the following all steps are summarized.

a) Select products from data dictionary: The aim of this step is to derive a small set of PAs to create common PLAs. Due to the huge number of products and their variants in the data dictionary, a selection is crucial for the creation of the initial APLA. A product is based on a software project. A software project defines the scope of modules, groups of modules, groups of groups (hierarchy) and interfaces reused for integration. The interface is described by modules and contains references to globally available variables. The required type and the provided type of references are distinguished. To realize a communication between two modules, it is necessary that one of the two modules provides the variable and the other consuming module requires the variable. We call this a dependency. Variables themselves store valuable data for the communication. A provided variable must also be declared (ANSI C like) and is therefore owned by the declaring module. PAs consist of modules, groups, and associated dependencies. All those elements have a set of data dictionary related attributes with a special meaning, which are considered to determine the initial selection of PAs. A problem arises when the exploration of extracted information is not manageable because of the big data set. Therefore we define selection criteria to extract a smaller set of PAs from the data dictionary. The following items describe examples for selection criteria in details:

- Projects and modules that have the release status. A project in release means that it is already integrated by TIER 1. A module in release means that it is already realized and positive tested by OEM.
- Modules that are referenced by selected projects.
- Projects that are related to one of the required engine control unit (ECU) generations.
- The most recent created modules and projects.

b) Recover PLA candidates using data mining: A very common approach to recover patterns and structures in large data sets is to use data mining methods and techniques. Many various techniques exist and are used in practice with different advantages and disadvantages for recovering an APLA. In this methodology we chose an approved approach, which provides good results in the field of recovery structures in information systems. The approach is called Spectral Analysis of Software Architecture (SPAA) [31], [32], [33] and is a generic approach to cluster software elements by their dependencies.

The SPAA approach is divided into three steps as visualized in Figure 5. First, all dependencies between all elements within the scope have to be identified. The type of dependency varies and depends on the kind of system, e.g., for object orientated information systems dependencies like classical call, extends, or implements relations are useful [32]. In the next step the constructed directed graph has to be weighted - the higher the edge weight value the lower the probability of cutting this edge in the clustering step. The weighted graph is clustered with a Spectral Clustering algorithm considering that this is a good heuristic to solve this NP-hard graph cut problem as described in [31] and [32].

As input data for the SPAA approach we choose all modules, which are contained in the selected products. Between these modules we determine dependencies depending on the provided and declared variables (see *step a)*). In this case the edge weight is defined as the sum of shared variables of the corresponding modules.

Often a heuristic is used to suggest the number of clusters. The preferred heuristic for Spectral Clustering is the eigengap heuristic, due to the fact that Spectral Clustering determines the eigenvalues of the normalized Laplacian, which are also used for this heuristic - described in detail in [31] and [32]. The application of Spectral Clustering results in a cluster separation of the weighted graph, as presented in [32] the modules can be clustered in a hierarchical way. Therefore the clusters have to be used as input data for the Spectral Clustering algorithm again. These procedure can be repeated with each generated cluster until the level of partitioning is satisfying. Summarizing, the elected data mining technique creates a PLA candidate of the selected products including a hierarchical grouping of modules and indicating the inter group dependencies.

c) Recover PLA candidates using functional analysis: The aim of this step is to recover a PLA candidate using a technique considering the functionality aspects. In the ECU software development most of them are open/closed control loop related functions [34], [35]. At first we create a number of processing function related groups, which are determined by expert knowledge. For each group a set of modules is referenced using the product scope. The references enable the tracing between PA elements and data dictionary modules. In the next step, the dependencies between the groups are created. Thereby only variables are considered that need to be shared between groups. The scope of other variables remains restricted. Some of the created groups may have a similar but more coarse grained function scope. Those can be again aggregated together, which leads to a hierarchical structure. Applying the above technique results in another PLA candidate, which consists of several hierarchical groups and group dependencies.

d) Integrate APLA from PLA candidates: The *steps b)*, *c)* produce PLA candidates by different recovery techniques. Instead of *steps b)* and *c)*, other techniques from the field of architecture recovery could be used. But exactly one APLA is required for the following managing activities (see Section III). Therefore the integration of all available PLA candidates is necessary. We propose two essential steps for integration: At first groups are created, which represent the leafs of the APLA. Therefore the appropriate groups of the PLA candidates are compared and evaluated for reuse. Next the dependencies between groups in the APLA are determined. In the second step the aggregation of the leaf groups is created reusing groups of the appropriate level from the PLA candidates. The resulting groups are determined again by a comparison in an evaluation step. The second step is applied iteratively for each available PLA candidate level.

e) Discover TPLA candidate from automotive domain knowledge: As an initial starting point for the following managing activities (see Section III-A) a TPLA is needed. A TPLA contains at least the planned structure compared to

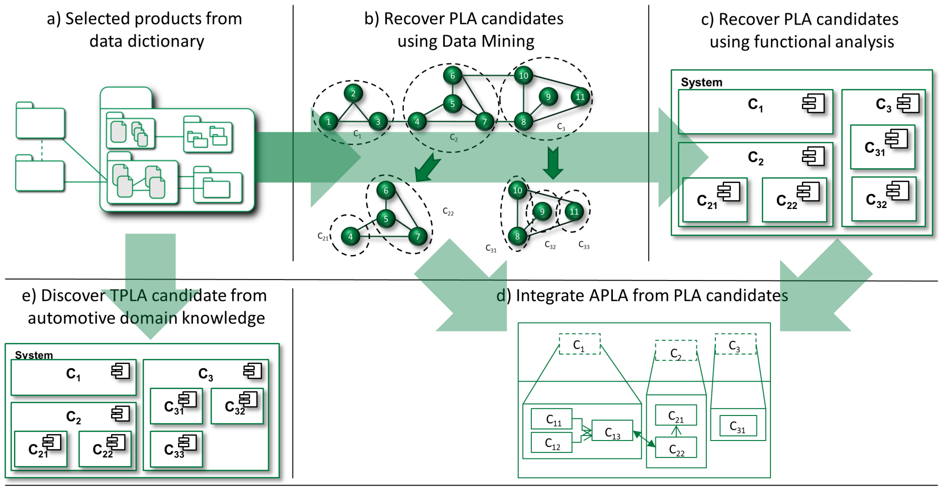
a) Selected products from data dictionary
b) Recover PLA candidates using Data Mining
c) Recover PLA candidates using functional analysis
d) Integrate APLA from PLA candidates
e) Discover TPLA candidate from automotive domain knowledge
System
C1
C2
C3
C21
C22
C31
C32
C33
C11
C12
C13

Figure 4. Overview of activity Recovery & Discovery

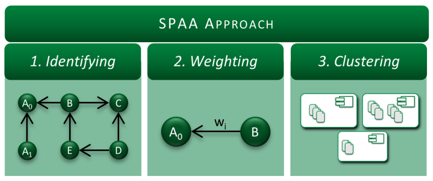

Figure 5. Overview: SPAA approach

the APLA. This knowledge has to be identified from product experts. As the architecture documentation is only available for individual projects, the knowledge for planned changes considering a PLA must be imposed using domain knowledge. To create the structure of a desired TPLA, group candidates and dependency candidates are identified from standardized automotive specific reference architectures [36], [37]. The TPLA is created iteratively considering the knowledge of experts.

B. Results from Real World Example

We have applied the methodology for extracting initial architectures on a real world example, the engine control unit software at Volkswagen. Next we show how we applied *step a)* to *step e)* to the example. We need to introduce the concept of the *function package* for further consideration. A function package references a set of modules or further function packages and serves for functional grouping.

a) Select products from data dictionary: As a starting point, we use the software repository of the engine control unit software at Volkswagen. The analysis was carried out in July 2015. At the time, the repository contained 21,734 versions of modules. First, the projects to be considered were selected. We wanted to consider a wide range of different projects. Thus, we have selected projects from two different suppliers and for different types of engines: From the first supplier a diesel and otto variant, respectively, and from the second supplier a diesel, otto, and otto-hybid variant. The following selection criteria were used to reduce the number of module versions: Only modules and function packages are selected

- that are not contained in a further function package,
- that are referenced by the selected projects,
- that have the release status, and
- that are the most recent created versions.

After applying the selection criteria, 162 modules and 43 function packages were selected.

b) Recover PLA candidates using data mining: We applied Spectral Clustering resulting in a cluster separation of the weighted graph. The procedure was repeated with each generated cluster until the level of partitioning was satisfying. A number of clusters with 16 or 18 clusters has turned out to be satisfying for all selected projects. Although the degree of cross-linking between the given modules is very small, the coupling between the clusters is relatively high. The cause of the high coupling may have various reasons, e.g., unsuitable parameterization or poor modularity.

c) Recover PLA candidates using functional analysis: From the study of the selected 43 function packages, abstract groups were identified by expert knowledge. Non-grouped elements are too complex for a manual, professional investigation. However, the generated groups have a high degree of interpretation (structural and technical).

d) Integrate APLA from PLA candidates: In this step, we first looked at the similarities and differences between the two PLA candidates from *step b)* and *step c)*. The aim was to derive a common APLA from the two PLA candidates. We built an integrated APLA for selected parts of the given PLA candidates. Here a lot of manual work is necessary. For the scope of the engine control software, the approach of *step d)* has ultimately not scaled. We could not provide the necessary manual work for building the integrated APLA for the entire engine control unit software with the two doctoral students working in the project.

e) Discover TPLA candidate from automotive domain knowledge: We analyzed several standardized automotive specific reference architectures [36], [37] to create a TPLA. There are recurring structures for combustion engines, for example: air system, fuel system, combustion model, etc.

Furthermore, there is a systematic structure of the hierarchies and dependencies, for example: driver's request, propulsion request on the power train, and power train units. We used this information to create a first draft of a TPLA for the engine control unit software. The TPLA is not specialized to a certain kind of engine like otto, diesel or hybrid. This draft TPLA was then discussed with experts from Volkswagen. Some minor adjustments were necessary until we had a final version of the TPLA.

Summary: By applying the proposed methodology, we could recover an APLA and discover a TPLA candidate. As shown in step d), however, difficulties have arisen in building an integrated APLA due to the size of the selected system. To handle such huge systems an automated process must be developed by further research. Even without performing the integration step, the two PLA candidates created are a useful basis for analyzing the current eroded system architecture. The essential structures could be made explicit by our approach.

The TPLA candidate and the APLA are then used in activity `PL-Design` and the subsequent activities (see Figure 2): The alignment of both PLAs is planned and implemented in order to repair the eroded architecture. Finally, the repaired architecture is further developed by the long term evolution cycle as described in Section III-A.

V. An Approach to Extract Concepts from System Realizations

For the specification of software architectures design patterns, architectural patterns or styles are an important and suitable means, also in other engineering disciplines [17]. We subsume these under the term of architectural concepts. An **architectural 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 constraints on how they can be combined [17]. To get a better understanding of the wide spectrum of architectural 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, ...

Based on this and our experiences made during the application of our approach described in the previous Section IV, the development of a new approach focusing the architectural concepts was part of the ongoing research activities.

In this section we will introduce this new approach with the aim to support the Recovery & Discovery activity. In Section VIII we will give an outlook on how this approach can be integrated into an evolutionary incremental development process and how a knowledge based process for architecture evolution and maintenance for architectural concepts can be implemented.

A. Introduction of the Approach

Based on the experience made during the practical project work, it became apparent in the Recovery & Discovery activity that an important issue to get a substantiated comprehending of a product architecture is to make not only the architecture explicit, but also the architectural concepts. Looking at different products and their architectures, the concepts are very helpful to create a common product line architecture. For this reason, the following research question (RQ) was focused in the ongoing research process: *"How can developers' best practice be identified and reflected to the architecture level?"* - From this general research question the following three research questions were derived:

RQ 1: How can a concept be represented with regard to

(a) composition to higher and usually unknown abstract concepts and

(b) the transferability of knowledge to or from other systems?

RQ 2: How can architectural concepts be algorithmically extracted and identified with regard to

(a) the large number of different concepts and

(b) their variations on different abstraction levels and contexts?

RQ 3: How can a tool support be realized with regard to

(a) false positive results respectively concept candidates and

(b) the huge amount of source code (scalability) ?

The outcome of this research activity is the approach shown in Figure 6. In this and the following section the research questions will be answered in detail.

We start with some general definitions. A **Concept** C is described by a set of **Properties** P. For an **Element** E a so called **Detector** D is defined as the binary function $d_{p_j} \in D$ for a concrete property $p_j \in P$ and a concrete element $e_i \in E$:

$$d_{p_j}(e_i) = \begin{cases} 1 & \text{, iff the Element } e_i \text{ fullfills the Property } p_j \\ 0 & \text{, otherwise} \end{cases} \tag{1}$$

An element can be a system artifact like a class, a function or a dependency between two elements as well as a subset of artifacts and their dependencies of the realized systems.

As shown in Figure 6 the input for the extraction cycle is the realization of the systems. The system artifacts respectively the source code elements are transferred to the so called **System Snapshot** $\mathfrak{S}$. It represents the realization of a software system as a language independent model representation, but including the links to the original source code elements. The used meta-model is a further development of the model used in [5], [38] and [39].

Another data pool is the **Factbase** $\mathfrak{F}$, which represents the fulfillment of concepts for the concrete elements. It is divided

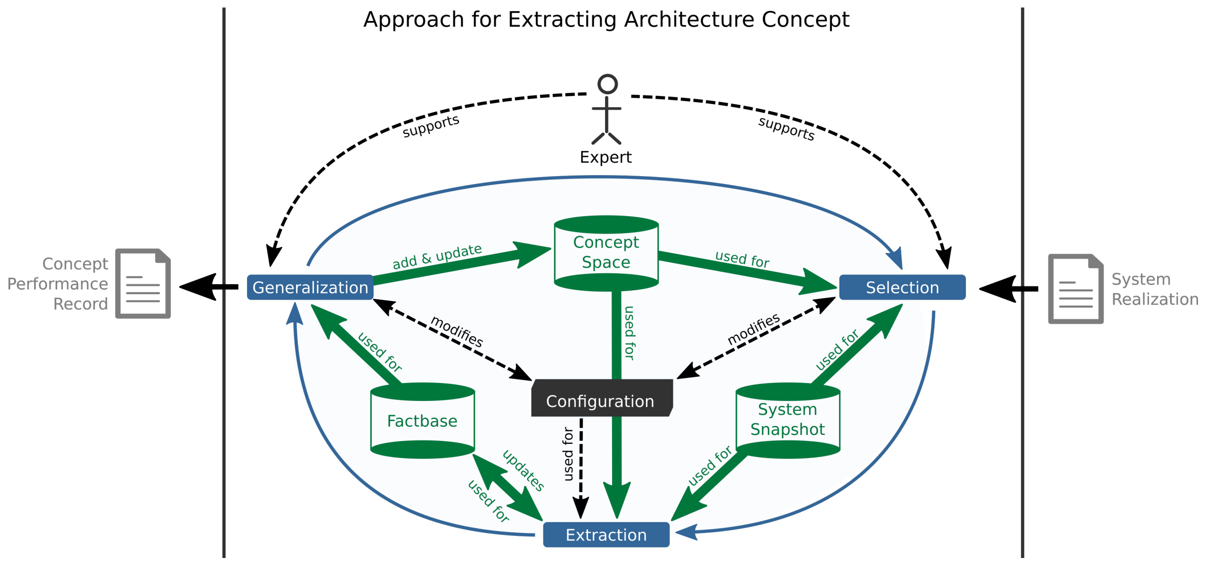

Figure 6. Approach to extract architectural concepts from system realizations

into three parts, two data-structures, which are organized in a table-structure, listing facts referring to elements respectively to dependencies and one data-structure describing facts about elements and the dependencies between them. These facts are organized in a graph-data-structure.

The last of the three data pools is the **Concept Space** Ω. It stores all known concepts, whereby a concept is represented as a named element and linked to its detector and examples, which fulfill this concept.

Altogether the defined process for extracting architectural concepts consists of three activities (blue boxes in Figure 6), which are performed iteratively and is called Extraction-Cycle. The connecting element between these activities is the **Configuration** Σ. Per iteration one configuration $\sigma_i \in \Sigma$ is created and used for the information exchange between the activities. Therefore, it includes all decisions that are made in an activity.

As shown during our studies it is not that difficult for an expert to decide, which system parts are relevant for a concrete analysis, as well as to validate if a concept is a ”real” concept or not. Because of this finding we integrate an expert to support two activities.

The output of the approach is the so called **Concept Performance Record**. This record informs about the concepts that are found in the analyzed system realization.

In the following the three activities are introduced in detail.

1) The Selection Activity: In this step an expert decides, which parts from the system should be analyzed and what is the initial set of concepts, which should be used for it. The expert will be supported by typical tools presented in Section IV like SPAA, to get a system decomposition, which is performed on the System Snapshot for example. On the other hand it is possible to reduce the number of concepts from the Concept Space, because of some basic knowledge of the system, like if it is object orientated or not.

The selected sub set of the system and the selected set of concepts, which are represented by its detectors, are stored in the Configuration and used as input for the next step.

2) The Extraction Activity: This step is fully automated and generates first the Factbase based for the selected elements by executing each detector for each element. Next different algorithms from the field of machine learning and neural computation are used, which are named in detail in the following Section V-B, to extract potential new facts and/or combinations of them. These so called **Concept Candidates** $\hat{C}$ are added as non-valid concepts to the Factbase. This includes also the **Representatives** R of this concept candidate thus elements, which fulfill this new extracted concept.

If this extension step is completed, the transition to the generalization step takes place.

3) The Generalization Activity: After the Factbase has been enriched with new facts receptively potential new concepts, a validation of these facts is carried out in this activity by an expert. The expert decides on the basis of the representatives of this concept candidate, whether the concept is a real concept or not. These decisions are stored also in the configuration. Thus the configuration includes the information about selected detectors and system artifacts and the concepts newly extracted and validated on this basis.

Based on this decision making process new detectors are generated. This can be done manually or automated by training a so called neural-network detector with the representatives in order to detect these concepts in the future. Finally, this new knowledge has to be integrated into the Concept Space, thereby it will be checked whether the new extracted concepts are already contained in the Concept Space and have not been selected in this iteration. In this case, the expert has to decide if it is the same or a different concept, i.e., whether it should be added as new or the already existing detector is re-trained with the new representatives.

Furthermore, the expert has to decide whether the Extraction Cylce should be terminated or a further iteration should be executed. In the next iteration new concepts can be extracted, which are based on concepts learned in the previous iteration, or by selecting other system artifacts.

B. Implementation of the Approach

In this subsection a concrete implementation of the proposed approach will be described. The chosen algorithms are not fixed for the individual steps and can be replaced by algorithms, which perform better. In the following the algorithms are listed, which are used to fulfill and support the individual actions.

Within the Selection activity the SPAA approach, illustrated in Figure 5, is used to create different views of the system decomposition to support the expert by selecting relevant elements and detectors.

For the extracting of new concept candidates within the Extraction activity different clustering algorithms and a statistical analysis were implemented and benchmarked. The input for all algorithms is the generated Factbase. Statistical analysis based on the frequency analysis of occurring patterns gave first indication for potential concept candidates but was not practicable for a good automation of the extraction process.

Therefore, different clustering algorithms were used to group similar elements and to derive concept candidates from this clusters: Neural Gas [40], Growing Neural Gas [41], and a Self-Organizing-Map (SOM) [42] orientated on the work of Matthias Reuter [43], [44]. These algorithms are used to find concepts on the system element level to detect special data-objects like TransferObjects [45], for example. In addition, they are used to extract similar properties for the dependencies between elements to define different types of dependencies like special communication channels or different kinds of relations like an inheritance relation between two elements, which is typical for an object orientated realization, for example.

To extract new facts from the facts represented in a graph-structure, we use the following algorithms to find similarities and anomalies within the graph:

1) Graph Kernels [46],
2) Graph Clustering approaches like SPAA [31], [32], [33], and
3) t-SNE [47].

For the creation of new detectors by training them with the representatives, a SOM is used. The selection, parametrization and evaluation of suitable algorithms are an ongoing process and will be focused in future work.

C. Supporting Recovery & Discovery

In the introduction of this section three research questions were derived, which can be answered by the presented solution. The answers can be summarized as follows:

RQ 1: (a) Concepts can be represented by their characteristic properties, whereby they can be organized in a hierarchical way, so it is possible to define higher respectively even more complex concepts. Because of detector mechanism and the possibility to create new detectors by training them with the representatives of this concept, it is possible to abstract from concrete instances. (b) So it is possible to check any element, also from other systems, if it is fulfilling a concrete concept or not.

RQ 2: (a) It is possible to define for each well known concept a detector, but it is the goal of this approach to find new concepts by clustering elements and extracting structures, which may represent a concept. So it is easier for the expert to decide if this is a kind of concept and what is the objective of it. (b) Because of the Concept Space it is possible to find similar concepts and also to merge them or to define explicit new variants of an existing concept maybe for different contexts, for example.

RQ 3: (a) Because of the validation step within the Generalization activity recommended concepts, which are false positive results, are marked as anti-concepts and they are also stored in the Concept Space, so in the next iteration they can be filtered. (b) To handle a huge amount of source code files a tool supported decomposition process of the system was integrated into the Selection activity. Furthermore, the extraction cycle was designed as an iterative process, to focus step by step on different aspects or parts of the system but take the so far extracted knowledge into account.

With the help of this approach it is possible to integrate the aspect of architectural concepts into the Recovery & Discovery activity. For example, the Coordinator concept, which will be introduced in detail in the following Section VI-A and illustrated in Figure 8, can be determined with the presented approach.

In Figure 7 an excerpt of the Factbase is visualized representing elements and their dependencies. The different properties for elements and edges can be mapped to colors, and the edge weights are summing up the number of dependencies of the same type. As a result of the Extraction step the blue-green nodes (see Figure 7) are recommended as concept candidates. During validation of this candidate the expert looked to the representatives of this candidate and determined that this element is only connected to green nodes, which are Filters, by only one instance of a communication dependency to each node, which allows only the transmission of state and mode information. We call an element with such characteristic properties a coordinator. So the extracted candidate is a valid concept and can be integrated into the Concept Space including the creation of a detector to have the possibility to check any element if it is fulfilling the coordinator concept or not in the future.

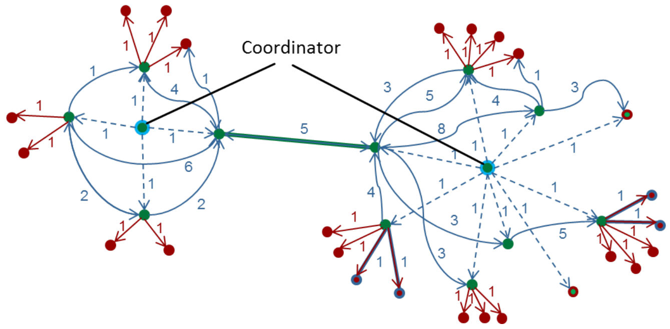

Figure 7. Excerpt of the Factbase

VI. Planning and Evolving Automotive Product Line Architectures

A. Concepts for Designing Automotive Product Line Architectures

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 [18].

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 explored for our field study. In the following we will present architectural concepts, which are addressing this problem in particular. Figures 8 and 9 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 has 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 is used, but this data can change over time, which 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 8.

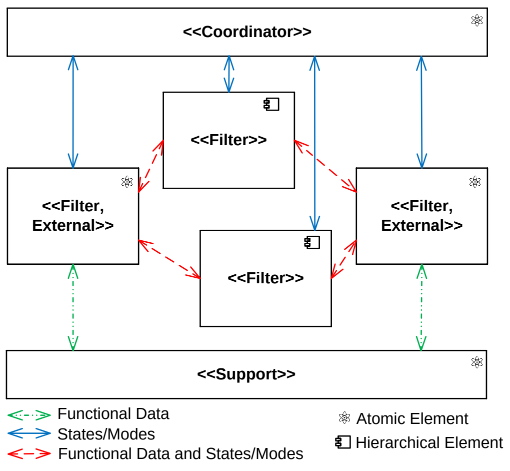

Figure 8. Architecture design principle: External elements

Each component has to declare a port for states and modes to uncouple the calculation of the system state from the component. The mode in which a component is currently located indicates the mode of execution of a certain function, like "kickdown", "emergency brake", "active", or"inactive", in the case of driving functions. 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 8 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 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 inhouse, some functions are implemented by external suppliers. But OEM components have requirements resulting in changes of interfaces, behavior or functionalities of theses externally 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 8 shows a simple solution to handle external elements: `Filter` components developed externally are annotated with `Filter, External`. By using this annotation, one can identify with little effort, which component is external, and which connections are affected.

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 9, a `Filter` component can be decomposable, a so called non-atomic component contains a structure, which follows the design principle visualized in Figure 8. It includes a `Coordinator` and `Support` component and an arbitrary number of `Filter` components. Whereby the inner `Filter` components have explicit defined responsibilities.

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 "Anvil-like 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 10, a component consists of two parts with different responsibilities - `Execution control` and `Function algorithms`. Each part has a defined set

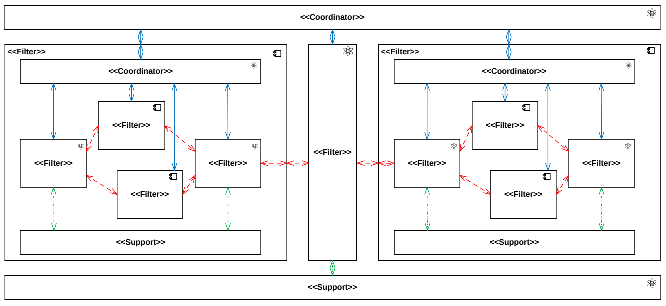

Figure 9. Architecture design principle: Hierarchical communication

of interfaces, types of communication channels, and exchange data. Due to the separation into two distinct areas, components are visualized as anvils (see Figure 10).

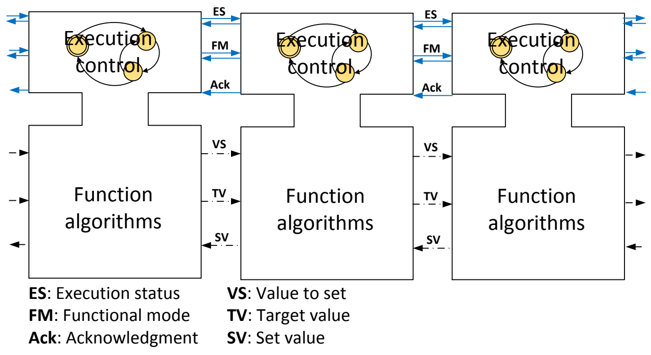

Figure 10. Architecture design principle: Anvil-like component model

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 value to set (VS) and a target value (TV). VS is the value to be set by the actuator in the next computing cycle, e.g., the new torque value for the next cycle. TV is the value, which is to be achieved in the future, e.g., the torque value requested by driver. To achieve TV a set of computing cycles is required. The set value (SV) of the function algorithms is the value that is currently set by the actuator and is transferred in the opposite direction compared to VS and TV. The aim of SV is to inform the components about the value currently set by the actuator. The functional algorithms exchange only functional data with one another.

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 functional dependencies, additional point-to-point connections are added, which we call "technical dependencies" between the components in the following. 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 11). 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 11 shows the architecture design principle *feedback channel.*

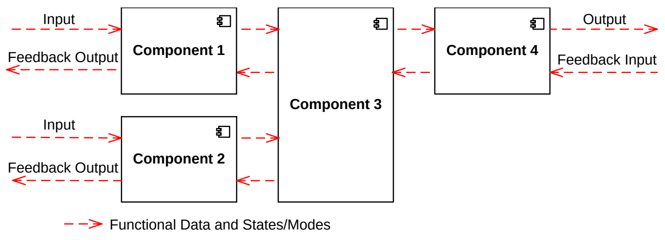

Figure 11. Architecture design principle: Feedback channel

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 11 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 to the current system state without the necessity to have an explicit connection to the corresponding 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 VI-A4 (Anvil-like Component Model, Figure 10), describes coordinating functionality, status and mode information and functional data connections. All these aspects we can find in the design principle VI-A2 (Coordinator, PipesAndFilters, Support, Figure 8), too. With the difference that the `Anvil-like Component Model` concept is for low level control functions, whereas the other concept deals with components on another abstraction level - to clarify the `Anvil-like Component Model` principle can be applied for a `Filter`, for example.

Architectural 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 [18].

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 [5]. 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% of earnings before interest and taxes [48]). 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 modules of products shall be reused for other products, a high reuse-rate on the product level 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 III-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:

$$\mathcal{E}^{PLA} = \frac{number\ of\ concerned\ LAE}{number\ of\ all\ LAE} \tag{2}$$

$$\mathcal{E}^{P} = \frac{number\ of\ concerned\ products}{number\ of\ all\ products} \tag{3}$$

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:

$$\mathcal{R}^m = \frac{number\ of\ usage\ of\ m\ in\ all\ products\ of\ pla_x}{number\ of\ all\ products\ of\ pla_x} \tag{4}$$

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

$$\mathcal{R}^M = \frac{\sum \mathcal{R}^m}{number\ of\ all\ modules} \tag{5}$$

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

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%).

TABLE III. Reuse rate for the example of Figure 3.

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

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 activity 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:

$$\mathcal{A}^p = \frac{number\ of\ all\ exports\ of\ all\ modules\ used\ in\ p}{number\ of\ all\ exports\ of\ all\ modules\ in\ p} \tag{6}$$

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

$$\mathcal{A}^P = \frac{\sum \mathcal{A}^p}{number\ of\ all\ products} \tag{7}$$

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

$$\mathcal{A}^{PLA} = \frac{number\ of\ all\ exports\ of\ modules\ used\ in\ all\ products}{number\ of\ all\ exports\ of\ all\ modules\ of\ all\ products} \tag{8}$$

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
$\mathcal{A}^{p1}$	$\frac{1+1+0}{3+4+2} \approx 0.22$	–	$\frac{2+0+0}{3+4+2} \approx 0.22$
$\mathcal{A}^{p2}$	$\frac{1+0}{3+4} \approx 0.14$	$\frac{1+0}{3+4} \approx 0.14$	$\frac{1+0}{3+4} \approx 0.14$
$\mathcal{A}^{p3}$	–	–	$\frac{1+0}{3+4} \approx 0.14$
$\mathcal{A}^{P}$	≈ 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 12 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.

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

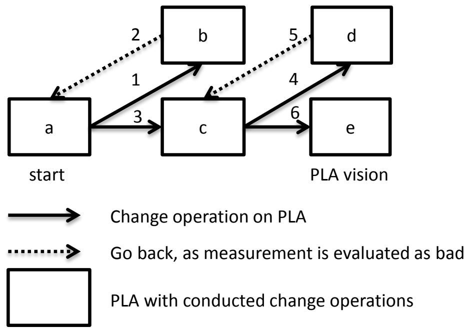
2
b
5
d
1
4
a
3
c
6
e
start
PLA vision
Change operation on PLA
Go back, as measurement is evaluated as bad
PLA with conducted change operations

Figure 12. Example: Applying change operations on a PLA

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.

VII. Case Study

In this section we introduce a real world example, a longitudinal dynamics torque coordination software, from automotive software engineering. We apply our methodology for planning and evolving automotive product line architectures on this example and present the results of a corresponding case study.

A. Real World Example: Longitudinal Dynamics Torque Coordination

Our approach for designing the logical architecture described in the previous sections 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.

In our example, we consider the control of the acceleration and braking process, which is controlled by the driver via the accelerator and brake 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 electronic stability control (ESP) or the adaptive cruise control (ACC), which can act as accelerator and decelerator. In the case of longitudinal dynamics torque coordination (see Figure 13), both acceleration and braking processes must be coordinated with one another since there are mutual interdependencies. A drive train coordinator (DTC) was introduced for the coordination of the acceleration path.

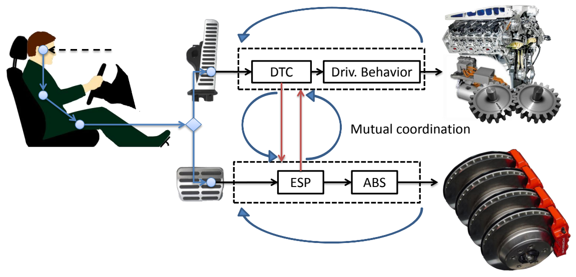

Figure 13. Automotive powertrain example: Mutual coordination

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 of the requesters was implemented in the example by the addition of a new explicit communication for the solution of coordination problems (see Figure 13, “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 in the software components. Furthermore, accidental complexity has increased disproportionately because of the wide interfaces and strong coupling within the architecture of the system.

Next, we describe how we applied the approaches introduced in the previous sections to manage the complexity of the example system. This paves the way for long-term maintenance and extensible architectures.

B. Origin of the Growing Accidental Complexity

In the following, the problems outlined above are explained in more detail using the real example. Later, we will show how by using different architectural principles, a significantly improved product line architecture with low accidental complexity can be build. This example is based on real industrial projects, but these have been simplified in this paper in order not to disclose business secrets.

The example consists of two systems that existed separately from each other in the past. The systems are, on the one hand, the acceleration path to the engine, where the driver generates a positive torque request to the engine by actuating the accelerator pedal. The other system is the braking path, on which the driver transmits a negative torque request, the so-called deceleration request, to the brake by actuating the brake. In both systems, the pedals were connected directly to the engine or the brake by a bowden cable.

With the development of increasingly better and more cost-effective electrotechnical systems, both systems have been further and further electrified. In the braking path, assistance systems were introduced to increase safety, such as the anti-lock braking system (ABS) and later an ESP. A control unit, the engine control unit, was introduced into the acceleration path, which led to the electric accelerator pedal in the 90s. This resulted in the elimination of the direct bowden cable to the engine. Furthermore, assistance systems have been developed, which optimally transfer the driver’s request torque. By introducing electric motors, it is now also possible to set negative torques on the drive path. Thus, it was necessary that both systems exchange information with each other. As a result, all systems had to be connected to each other in order to be able to match the desired values with the real values of the motor and brake, respectively.

Architecture recovery: As outlined above, further developments have led to an erosion of the originally planned architecture. The implementations in the individual products have increasingly deviated from the planned product line architecture. Finally, the existing system was very difficult to handle in further developments. For this reason, the system had to be revised and, in particular, the architecture had to be repaired. Thus, we applied architecture recovery as described in Section IV. Recovery uses reverse engineering techniques to extract the implemented architecture from source artifacts. Figure 14 illustrates the recovered architecture. The black arrows show the data flow along the two paths. In addition to the physical set values, this data also contains information about the state of the assistance system as well as the information about the mode (kickdown, emergency brake, active, inactive, etc.), in which it is currently located. The blue arrows convey the changes of the values for the torque requests. These connections are required, since there are controllers in all systems, which are integrated over time if the behavior of their control loop is not known.

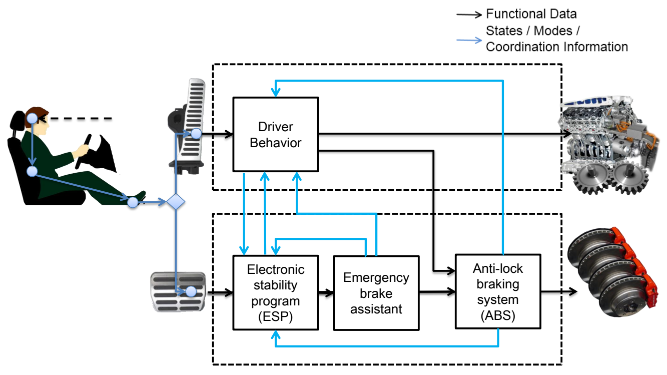
Functional Data
States / Modes / Coordination Information
Driver Behavior
Electronic stability program (ESP)
Emergency brake assistant
Anti-lock braking system (ABS)

Figure 14. Automotive powertrain example: Recovered architecture

C. Applying our Approach on the Example

Design of the new PLA - Iteration 1: After the analysis of the system, it became clear that the coordination information had to be reduced. The first step was the introduction of a coordinator component with the architecture design principle *Coordinator-PipesAndFilters-Support*, which enabled the coordination of both torque paths (acceleration/brake). The result of this change is depicted in Figure 15.

Measuring of architecture quality - Iteration 1: The data flow was not changed by the introduction of the coordinator, s.t. the functional behavior of the assistance systems remained unchanged. However, many interfaces necessary for the coordination information could be removed. This has reduced the complexity of many assistance systems. The complexity of the ESP, e.g., could be reduced to the level of the essential complexity. However, the ABS assistance system has risen in complexity since a further interface for the coordination had to be added here. The coordinator is also a very complex system since the coordinator now contains the entire coordination effort, which was previously distributed to the individual assistance systems.

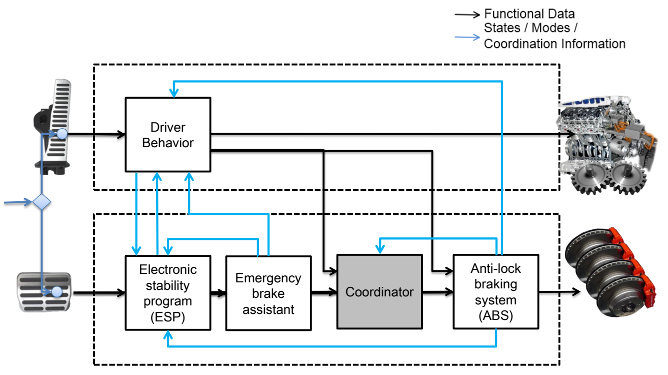
Functional Data
States / Modes / Coordination Information
Driver Behavior
Electronic stability program (ESP)
Emergency brake assistant
Coordinator
Anti-lock braking system (ABS)

Figure 15. Reducing complexity by architecture design principle *coordinator*

Design of the new PLA - Iteration 2: In order to ensure that no additional coordination information interfaces are generated, the architecture design principle *feedback channel* (see Section VI-A5) is introduced for all components in the system. This ensures that all controllers are informed of the current situation in the system, without the need for additional interfaces to all components. The feedback interface has only to be added to the ESP assistance system. To optimize the information processing in the individual components, the architecture design principle *anvil-like component model* was introduced. Due to the division into the execution control and functional algorithms, the components became much more structured and readable. In the part functional algorithms, only all the technical complexities, which concern the function itself are contained. The part of the execution control contains all relevant system-dependent contents. This facilitates the development of each individual component, since adaptations, which have to be carried out solely because of a system change, only take place in the execution control. All tests regarding the functionality of the component can usually be adopted unchanged, since the functionality is implemented exclusively in the functional algorithms. The resulting product line architecture is shown in Figure 16.

Measuring of architecture quality - Iteration 2: By the new design the *modification effort* of a PLA could be improved significantly with regard to further development. If, e.g., a new assistance function is to be introduced, only few adaptations to the existing architecture are necessary. The evaluation of *cohesion* and *reuse rate* according to Section VI-B can not be carried out at this point because currently only a prototypical product version exists. It is, however, to be expected that the reduction of mutual interdependencies will lead to a significant increase in cohesion. In addition, the current implementation of the modules includes a high degree of variability, which increases the reusability in different products. Furthermore, the improved modification effort also contributes to an increased reusability over several subsequent product versions since adjustments are only necessary in a few places when new functions are introduced.

Planning of development iterations and prototyping: As shown in the example, the development was carried out in two iterations. Both iterations resulted in an executable prototype, which was tested extensively. The functionality was tested by means of the tool Time Partition Testing (TPT). TPT suits particularly well due to the ability to describe continuous behavior [49]. As a starting point for the tests, a simple environment model was created. The module and composite tests were carried out taking into account previously defined scenarios. The signals were then evaluated and compared with the scenarios.

As a result of the tests, neither errors were found in the module tests nor in the composite tests. The case study has demonstrated that the migration of the existing functionality into an improved architecture is possible by means of our approach.

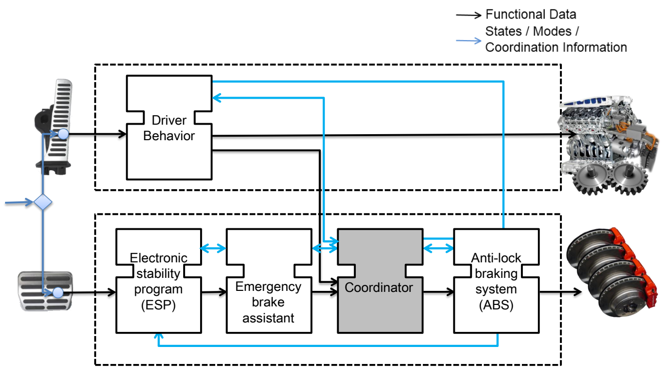
Functional Data
States / Modes / Coordination Information
Driver Behavior
Electronic stability program (ESP)
Emergency brake assistant
Coordinator
Anti-lock braking system (ABS)

Figure 16. Reducing complexity by architecture design principle *feedback channel* and *anvil-like component model*

VIII. Knowledge-Based Architecture Evolution and Maintenance

In this section we will outline how the approach introduced in Section V can be extended to a holistic solution for managing architectural concepts during the evolution of the system life-cycle. As visualized in Figure 17 the approach can be embedded into an evolutionary incremental development process. After each implementation step the realization can be analyzed.

Thereby the generated Concept Performance Record can support the system architect to get a comprehension of the

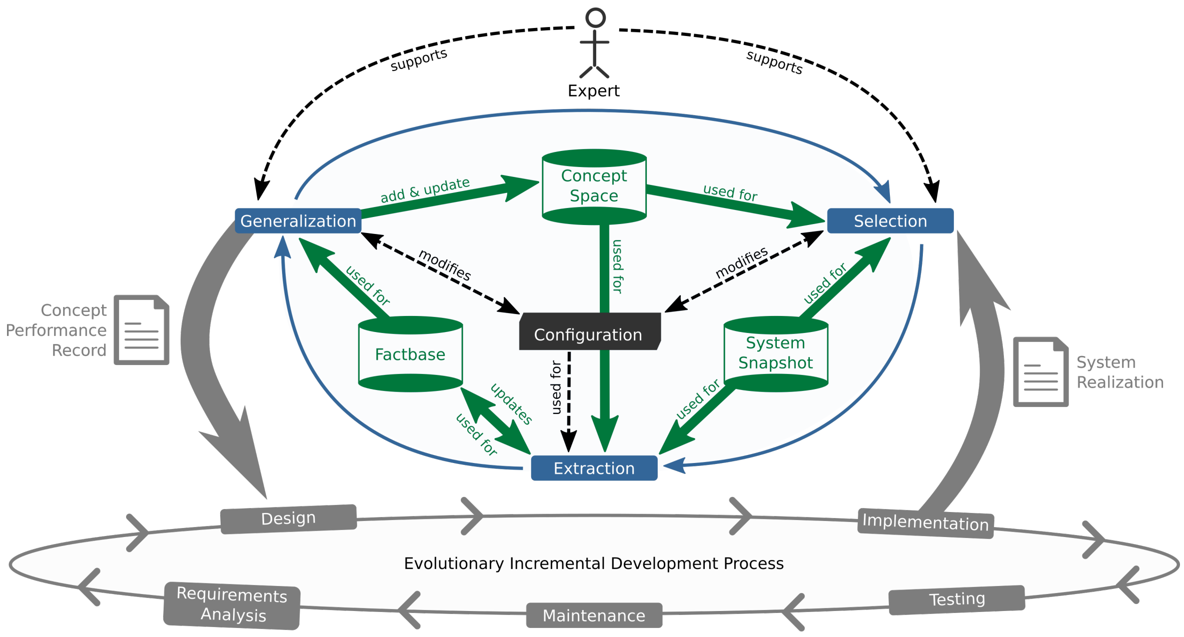

Figure 17. Overview of the approach to extract architectural concepts embedded into an evolutionary incremental development process

realized concepts. This information can be combined with the results from the `PL-Check` and `P-Check` activity (see Section III). As described the aim of the checking activities is to reduce the erosion of a product architecture by architecture conformance checking. The output of these activities are a list of violations. If the developer was not familiar with the architecture, for example, and this is the reason for the violation, it can still be fixed during the next implementation step by the developer, so that no erosion occurs. On the other hand it can be decided that the reason for the violation is reasoned by a not suitable architecture. In this case the Concept Performance Record can support by planning the architectural changes by making the aspects the developer has in mind explicit on the architectural abstraction level.

An additional issue is the improvement of the evolution and maintenance process by the monitoring of concepts. We can assume that the configuration and all data pools are stored in a repository and will be versioned. So we can answer the question: ”What might happen with architectural concepts over time?” - they can be adapted to new requirements or in consequence of new technologies, frameworks or programming paradigms, for example. This can also lead to new concepts, which maybe replace old concepts, so it might be possible that extracted concepts will disappear over time. But these changes can be detected with the help of the detector mechanism, too, or in other words comparing two Concept Performance Records from different versions of a product will lead to indications of mutations and/or displacement of concepts. What on the other hand can help to detect product architecture erosion at an early stage.

IX. Conclusion

We introduced a sophisticated approach for extracting, designing and managing architectural concepts and thus enabling long-term evolution of automotive software systems. The approach aims to close the gap between product architectures and the product line architecture in the automotive domain. Thus, we used adapted concepts like architecture design principles, architecture compliance checking, and further development scheduling with specific adaptations to the automotive domain.

With a high degree of erosion, a further development of the software is only possible at great effort. Before approaches to minimize erosion can be applied, the architecture must first be repaired. Thus, we investigated how approaches for architecture extraction can be adapted to be applied to automotive software product line architectures. First, we proposed methods used to extract initial architectures. Next, we explained our experiences gained from a real world case study. In the case study, we could recover a PLA for the engine control unit software. However, difficulties have arisen in building an integrated PLA due to the size of the selected system. To handle such huge systems an automated process must be developed by further research.

Furthermore, we integrated this recovery/discovery approach into an evolutionary incremental development process. We focused on how the developers best practice can be identified and reflected to the architecture level. In addition, we showed how a knowledge based process for architecture evolution and maintenance for architectural concepts can be implemented.

Next, we proposed methods and concepts to create adequate architectures with the help of abstract principles, patterns, and description techniques. Such techniques allow making complexity manageable. We presented architectural concepts developed within different industrial projects in the automotive domain involving different software architects and project members. For example, we introduced feedback channels enabling the dissolution of functional dependencies without the introduction of technical point-to-point connection.

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 demonstrated our concepts by an industrial case study from the automotive domain. We described how we applied the approaches introduced in the previous sections to manage the complexity of the example system. We have shown that the application of the approach paves the way for long-term maintenance and extensible architectures.

References

[1] A. Grewe, C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "Automotive Software Systems Evolution: Planning and Evolving Product Line Architectures," in Special Track: Managed Adaptive Automotive Product Line Development (MAAPL), along with ADAPTIVE 2017. IARIA XPS Press, 2017, pp. 53–62.

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

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

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

[20] 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.

[21] 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.

[22] R. Flores, C. Krueger, and P. Clements, "Mega-scale Product Line Engineering at General Motors," in Proceedings of the 16th International Software Product Line Conference - Volume 1, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 259–268.

[23] 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. Washington, DC, USA: IEEE Computer Society, 2008, pp. 187–194.

[24] L. Passos, K. Czarnecki, S. Apel, A. Wasowski, C. Kästner, and J. Guo, "Feature-oriented Software Evolution," in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '13. New York, NY, USA: ACM, 2013, pp. 17:1–17:8.

[25] Gentzane Aldekoa and Salvador Trujillo and Goiuria Sagardui Mendieta and Oscar Daz, "Quantifying Maintainability in Feature Oriented Product Lines," in Proceedings of the 12th European Conference on Software Maintenance and Reengineering. IEEE, 2008, pp. 243–247.

[26] Zhang, T. and Deng, L. and Wu, J. and Zhou, Q. and Ma, C., "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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] M. Schindler, "Automatische Identifikation und Optimierung von Komponentenstrukturen in Softwaresystemen," Master's thesis, TU Clausthal, 2010.

[32] M. Schindler, C. Deiters, and A. Rausch, "Using Spectral Clustering to Automate Identification and Optimization of Component Structures," in Proceedings of 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013, pp. 14–20.

[33] M. Schindler, A. Rausch, and O. Fox, "Clustering Source Code Elements by Semantic Similarity Using Wikipedia," in Proceedings of 4th Intern. Workshop on Realizing Artificial Intelligence Synergies in Softw. Eng. (RAISE), 2015, pp. 13–18.

[34] M. Körner, S. Herold, and A. Rausch, "Composition of Applications Based on Software Product Lines Using Architecture Fragments and Component Sets," in Proceedings of the WICSA 2014 Companion Volume, ser. WICSA '14 Companion. New York, NY, USA: ACM, 2014, pp. 13:1–13:4.

[35] D. Claraz, S. Kuntz, U. Margull, M. Niemetz, and G. Wirrer, "Deterministic Execution Sequence in Component Based Multi-Contributor Powertrain Control Systems," in Embedded Real Time Software and Systems Conference, 2012, pp. 1–7.

[36] K. Reif, Automobilelektronik - Eine Einführung für Ingenieure, 4th ed. Vieweg + Teubner, 2012.

[37] R. Isermann, Ed., Elektronisches Management motorischer Fahrzeugantriebe, 4th ed. Vieweg + Teubner, 2010.

[38] C. Deiters, Beschreibung und konsistente Komposition von Bausteinen für den Architekturentwurf von Softwaresystemen, 1st ed., ser. SSE-Dissertation. München: Dr. Hut, 2015, vol. 11.

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

[40] M. Cottrell, B. Hammer, A. Hasenfuß, and T. Villmann, "Batch and median neural gas," Neural Networks, vol. 19, no. 6, 2006, pp. 762–771.

[41] B. Fritzke, "A Growing Neural Gas Network Learns Topologies," in Proceedings of the 7th International Conference on Neural Information Processing Systems, ser. NIPS'94. Cambridge, MA, USA: MIT Press, 1994, pp. 625–632.

[42] T. Kohonen, "The self-organizing map," Neurocomputing, vol. 21, no. 1, 1998, pp. 1–6.

[43] M. Reuter and H. H. Tadijne, "Computing with Activities III: Chunking and Aspect Integration of Complex Situations by a New Kind of Kohonen Map with WHU-Structures (WHU-SOMs)," in Proceedings of IFSA2005, Y. Liu, G. Chen, and M. Ying, Eds. Springer, 2005, pp. 1410–1413.

[44] M. Reuter, "Computing with Activities V. Experimental Proof of the Stability of Closed Self Organizing Maps (gSOMs) and the Potential Formulation of Neural Nets," in Proceedings World Automation Congress (ISSCI 2008). TSI, 2008.

[45] A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, Eds., The Common Component Modeling Example: Comparing Software Component Models. Springer, 2008, vol. 5153.

[46] A. Gisbrecht, W. Lueks, B. Mokbel, and B. Hammer, "Out-of-Sample Kernel Extensions for Nonparametric Dimensionality Reduction," in Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), vol. 2012, 2012, pp. 531–536.

[47] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," Journal of Machine Learning Research, vol. 9, no. Nov, 2008, pp. 2579–2605.

[48] M. Bernard, C. Buckl, V. Döricht, F. M., L. Fiege, H. von Grolman, N. Ivandic, C. Janello, C. Klein, K.-J. Kuhn, C. Patzlaff, B. C. Riedl, B. Schätz, and C. Stanek, Mehr Software (im) Wagen: Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft. fortiss GmbH, 2011.

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