TU Clausthal

Dr. Christoph Knieke

Managed Evolution of Automotive Software Product Line Architectures

Cumulative Habilitation Thesis

ISSE-Habilitation 20

Institute for Software and Systems Engineering
Lehrstuhl von Prof. Dr. Andreas Rausch

Christoph Knieke

Managed Evolution of Automotive Software Product Line Architectures

Cumulative Habilitation Thesis

approved by the
Faculty of Mathematics/Computer Science and Mechanical Engineering
Technische Universität Clausthal, Germany
2019

Dedicated to my wife Dorothee
who always encouraged and supported me
in my scientific career.

Preface

In 2011, I received my PhD in Computer Science from the TU Braunschweig in Germany. Since December 2010, I have worked as a researcher and manager at the TU Clausthal. In late 2011 the research association Institute for Applied Software Systems Engineering (IPSSE) was founded as a cooperation between the TU Clausthal, the TU Braunschweig and the Volkswagen AG. I played a key role in the establishment and founding of the IPSSE and I am still working there as division manager and post-doctoral researcher. The focus of IPSSE lies in methods and tools for the development of embedded software. In this scope, one of the prevailing themes of IPSSE is the application of model-driven approaches to automotive software engineering. I have led numerous industrial projects since IPSSE was founded. In the first two years of my work at the TU Clausthal, I developed engine control unit software for series production on behalf of Volkswagen and gained valuable experience for my further research activities. In particular, I was able to identify relevant research questions, which I addressed in my thesis. At the end of 2015 several research areas were defined at IPSSE. I have led the research area knowledge based architectures and development processes and was responsible for several projects in this area and a group of PhD students. Since 2012, I have been working on methodologies and techniques for managing and developing software product line architectures in the automotive domain. This cumulative habilitation thesis embodies a summary of my work and a number of papers in this research topic.

Clausthal-Zellerfeld, October 2017

Christoph Knieke

Acknowledgement. I would like to thank my supervisor, Prof. Dr. Andreas Rausch, for the patient guidance, encouragement and advice he has provided throughout my time as post-doctoral researcher. My special thanks go to the PhD students of the architecture research group at IPSSE. These are Axel Grewe, Marco Körner, Andreas Rausch, Mirco Schindler, Arthur Strasser, and Martin Vogel. I thank all co-authors of the papers of this cumulative thesis. Last but not least, my thanks go to our collaborators at Volkswagen. Without the insight gained in numerous discussions and projects, this thesis would never have been possible.

Contents

Part I

Summary

1 Introduction

1.1 Automotive Software Engineering

The automobile has become the technically most complex consumer product [1]. The fulfillment of increasing customer requirements and strict legal requirements with regard to the reduction of fuel consumption and pollutant emissions, as well as the higher demands on safety and new driver assistance systems resulted in a steady increase in the deployment of onboard electronics systems and software.

The amount of software in cars has been growing exponentially since the early 1970s, and one can expect this trend to continue [2]. An important accelerator of this trend will be the introduction of automated and connected driving in the near future [3]. Software-intensive systems and functions are the major drivers for innovations in cars today [4]. In premium vehicles software is responsible for up to 80% of the innovations [5]. Up to 40% of the production costs of a car are due to electronics and software [2].

The requirements for automotive electronics differ significantly from other areas of consumer electronics. The following special requirements and characteristics have to be considered in particular (cf. [6], [7]):

- **Hard real-time requirements vs. limited resources**: Automotive functions have to have predictable behavior under all circumstances and have to react at the specified time. At the same time, the storage and computational power on ECUs has to be kept as small as possible in order to limit the costs as electric control units (ECUs) are built into a large number of cars.
- **High reliability and safety/security requirements vs. heterogeneity of domain knowledge**: Systems in an automobile must be reliable, safe and secure in all situations as system failures can endanger human life. The system's behavior depends to a large extend on the behavior of software functions. Experts from different domains such as electrical, control, and mechanical engineering are involved in the development of automotive functions. A lot of their domain knowledge is integrated into the software, but is often not explicitly available. Under these circumstances, ensuring a high degree of reliability, safety and security of systems is a difficult task.
- **Long product life cycles vs. short development cycles**: The OEM has the duty to offer service and spare parts for at least 15 years after the purchase of a vehicle due to the long product life cycles. In contrast, software is changed at comparatively short intervals. During the production period and even during the development phase, many new versions of a piece of software are developed. This trend will be accelerated in the future by the introduction

of connected services and new business models [8]. As a consequence of short innovation and long life cycles a huge number of versions and configurations exist, which inter alia makes maintenance very difficult.

- **Software development as product line vs. software sharing**: The high time and cost pressure in the automotive market encourages reuse of components and software in different vehicle projects leading to a high degree of variability within the software. Often, a product line approach is used to handle variability. In addition, OEMs want to reduce costs by reusing software in different engine control units of several suppliers. By applying software sharing, code relevant artifacts are delivered from the OEM to the Tier1 (supplier of OEM), or the other way around, from Tier1 to the OEM [9]. Each software from OEM and from Tier1 is developed orthogonally to each other as a separate product line and has to be combined and configured adequately for building a product.

Moreover, automotive software systems shift towards cyber physical automotive systems: The vehicles are connected - with each other, with objects in the surroundings and increasingly also with external information systems or the mobile devices of the driver and other involved persons [10]. These cyber physical automotive systems are based on new system architectures and development tools that address the complexity and enable the implementation and exploitation of massive amounts of interconnected devices and software services as well as embedded software in physical objects at different locations [10]. Under these conditions, the implementation of the numerous requirements for electronic systems of vehicles represents a development task of a high degree of difficulty [1].

The rapidly growing number of software-based features in the automotive domain asks for dedicated engineering approaches, models, and processes [4]. As key research challenges, Pretschner et al. [7] identify the integration of heterogeneous subsystems from different sources as well as their evolution and maintenance, and reuse.

1.2 Software Product Line Architectures

Today automotive manufacturers and suppliers design and implement complex applications by mechanisms that allow them to implement such functionality on integrated platforms. This offers the opportunity to build a variety of similar systems with a minimum of technical diversity and thus allows for strategic reuse of components. This has resulted in a growing interest in software product line approaches in the automotive systems domain [11].

Since the 1990s software product lines have been introduced as a major addition to existing reuse approaches [12, 13, 14, 15]. Clements et al. [15] define a software product line (SPL) as a family of systems that share a common set of core technical assets, with preplanned extensions and variations to address the needs of specific customers or market segments.

In general, software product line engineering consists of two key processes, domain engineering and application engineering [16]. The aim of the domain engineering process is to define and realize the commonality and the variability of the software product line. The process of application engineering is responsible for deriving product line applications from the platform established in domain engineering by exploiting the variability of the software product line [16].

Thiel et al. [11] present some challenges that automotive software engineering faces today and discuss contributions SPL approaches could make to provide solutions for these challenges:

- **Reduction of complexity**: Automotive software platforms are typically developed in such a way that they can be customized and used in hundreds of products. These platforms can easily incorporate thousands of variation points and configuration parameters. Managing this amount of variability is extremely complex and requires sophisticated modeling techniques.

- **Improved architectural design practices**: Software architectures for automotive systems need to be comprehensive enough to capture and describe the multi-functionality and all related issues.
- **Improved evidence in modeling and evaluating quality attributes**: A quality attribute is a non-functional requirement of a software intensive system, e.g., reliability, modifiability, performance, usability and so forth. Like any other domain, software quality is fundamental to any automotive system's success. However, quality as a concept is very challenging to define, describe and understand.
- **Design and evaluating architectures for family of systems**: Architecture for a family of systems helps identify the commonality among different systems and explicitly document variability.
- **Architecture knowledge management**: The knowledge required to make suitable architectural choices is broad, complex, and evolving, and can be beyond the capabilities of any single architect.
- **Increased process efficiency**: The current usage of models in automotive systems engineering is insufficient and is far from realizing its full potentials. For instance, models are used in isolated areas, without an integrated flow of information.

1.3 Life-Cycle Management and Managed Architecture Evolution

Software systems undergo continuing changes. Belady and Lehman [17] termed this dynamic behavior of software systems *evolution* and carried out empirical research on about 20 releases of the OS/360 operating system. The investigation led to five "laws" of software evolution: Continuing Change; Increasing Complexity; The Fundamental Law of Program Evolution; Conservation of Organizational Stability; and Conservation of Familiarity.

Software systems vary significantly in how easily they can be evolved to remain productive within a changing environment [18]. Cook et al. [18] call this quality of software systems "evolvability", defined as "*the capability of software products to be evolved to continue to serve its customer in a cost effective way*". Hence, software evolvability is an attribute that describes the software system's capability to accommodate changes. Rowe et al. [19] give a more profound definition of software evolvability, which also includes the term of architecture: "*An attribute that bears on the ability of a system to accommodate change in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity.*"

In developing long-life systems, evolvability must be ensured throughout the whole life cycle. Only by this way can software development be carried out efficiently and reliably and the productive lifespan of the software systems can be prolonged [20]. The possibilities for implementing changes are mainly determined by the software architecture [21]. Thus, software architecture evolution becomes a critical part of the software life cycle [20]. To enable software architecture evolution in the long term, architecture erosion and the growing of "accidental" complexity has to be minimized as explained in the following:

Architecture erosion: An important challenge with regard to the architecture is to minimize architecture erosion. In [22], de Silva and Balasubramaniam define architecture erosion as "*the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture.*" In our product line based approach, the product line architecture (PLA) constitutes the *intended architecture* whereas the architecture of the corresponding products of the product line constitutes the *implemented architecture*. The PLA is designed initially and

develops over time. It makes no difference whether the PLA is explicitly planned or exists only implicitly in the minds of the participants. In the further development, it must be ensured that the product architecture remains compliant with the PLA. However, due to the high time and cost pressure in the automotive sector, it is not possible for every further development to be controlled via the product line. Rather, some product-specific adjustments have to be made. This can lead (intentionally or unintentionally) to a product architecture that differs in comparison to the PLA: the architecture erodes. In the long term, this leads to reduced reusability and extensibility of the software artifacts.
Growing of "accidental" complexity: In the evolutionary development of automotive software systems, the range of functionalities grows steadily. Thus, the "essential" complexity of the product line architecture increases continuously due to the growth of the number of functions. However, the "accidental" complexity of the architecture of automotive software systems grows disproportionately to the essential complexity as illustrated in Figure 1.1 [23]. 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.

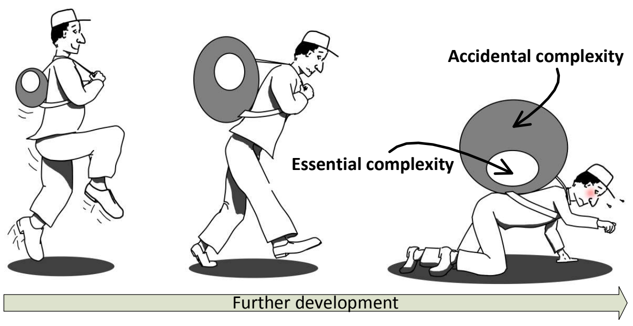


Fig. 1.1. "Essential" vs. "Accidental" complexity

Further challenges with regard to evolvability are the huge number of versions and configurations and the high degree of variability within the software. In addition, the parallel software product line development at OEM and Tier1 constitutes a great challenge with regard to evolvability due to the increasing complexity of the integration tasks.

In the following paper, we propose an approach for managed evolution of automotive software systems:

Andreas Rausch, Oliver Brox, Axel Grewe, Marcel Ibe, Stefanie Jauns-Seyfried, Christoph Knieke, Marco Körner, Steffen Küpper, Malte Mauritz, Henrik Peters, Arthur Strasser, Martin Vogel and Norbert Weiss. 2014. Managed and Continuous Evolution of Dependable Automotive Software Systems. In *Proceedings of the 10th Symposium on Automotive Powertrain Control Systems*, pp. 15–51.

In this paper we have identified three main challenges to strengthen automotive software systems engineering for the upcoming (r)evolution:

- **Complexity of automotive software systems and engineering processes has still to be manageable**: 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 development of vehicles cost efficient, modular components with a high reuse rate cross different types of vehicles are required. Thus, approaches from software product line engineering adapted to the special demands of the automotive domain are required. With respect to innovative and sophisticated functions, coming with the connected car and automated respectively 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 within the software product line.

- **Flexibility of automotive software systems and engineering processes has still to be provided**: Developing a new vehicle takes in average about 4 years. Commodity software used for vehicles, like operating systems, multimedia and infotainment software, or network drivers, is updated up to five times faster during vehicle development. This relation is even worse during vehicle operation. Customers expect that new functionality can be easily integrated into vehicles in a plug & play manner. However, nowadays integration of new hardware and software is very expensive. The adaptation of existing components is complex and error-prone. In order to respond quickly to these requirements, the development process must provide a high degree of flexibility.
- **Dependability of automotive software systems and engineering processes has still to be guaranteed**: Although a high flexible development process and a high reuse rate of automotive software components cross all vehicle types and variants are required, it is crucial to guarantee a high degree of dependability. As dependability, we summarize the essential software quality attributes for vehicles like availability, reliability, maintainability, safety, and security. Due to the connected car and automated respectively autonomous driving, safety and security becomes more and more critical. The actual high warranty costs of about 15% to 20% from earnings before interest and taxes caused by errors in automotive software have to be reduced. Guaranteeing dependability is a great challenge during development and operation of automotive software systems.

To cope with these challenges, we have sketched out an improved and sophisticated engineering approach for automotive software systems: Our engineering approach enables a managed and continuous evolution of dependable automotive software systems based on software product lines. It helps engineers to manage system complexity based on continuous engineering processes to iteratively evolve automotive software systems and the corresponding software product lines, and thereby guarantees the required dependability issues. To guarantee dependability, we have demonstrated a continuous, test-driven development and operating approach. We have shown how software product lines can support the derivation of test cases for the complete development cycle.

1.4 Contribution

After introducing the fields I address in this thesis, I give a summary of my research objectives. Afterwards, I will introduce a conceptual model with activities for managed evolution of automotive software product line architectures, and I will explain the contributions to the research objectives.

Research objectives:

1. **Recovery and discovery of automotive software product line architectures**: Software architecture erosion often occurs during further development of software. With a high degree of architecture 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. Architecture repair typically involves the two approaches *recovery* and *discovery*. Recovery uses reverse engineering techniques to extract the implemented architecture from source artifacts, and discovery hypothesizes its intended architecture [22]. Thus, we investigate how these approaches can be adapted to recover the implemented automotive software product line architecture from the developed products and to discover the intended product line architecture.
2. **Holistic approach for managed evolution of automotive software product line architectures**: In the automotive sector, it is not possible to carry out all further developments within the product line. Rather, there may be further developments that do not take place in the product line but at the level of the individual products. The reasons for this may be the high time and cost pressure, but also the fact that sophisticated further developments are initially to be tested within the scope of a prototype implementation. These further developments, which are separate from the product line, have to be transferred into product line development at a later stage. To this end, a holistic approach must be developed that adequately supports the two levels of development – product line and product – and their interaction. We define the following sub-objectives in our holistic approach addressing the architecture level (*a.*), and the software component level (*b.*) in automotive software product line development, respectively:
   a. **Maintaining stability of the PLA and minimizing software product architecture erosion in real world automotive systems even if extensive further development of the system takes place:** The PLA should be based on appropriate design principles that allow further developments with a minimal adjustment effort of the PLA. At the same time, software product architecture erosion is to be minimized. Thus, we use adapted approaches from architecture conformance checking in our approach.
   b. **High degree of reusability in real world automotive software development by achieving a high scalability, and a high degree of usage of the software components:** The reuse of software components is achieved by a product-wide development for different vehicle variants, as well as by reuse in subsequent products. However, the high variability within the software components increases the complexity of the components and thus makes reuse more difficult. From one vehicle generation to the next, functionality differs mostly not more than 10%, while significantly more than 10% of the software is re-written [7]. Thus, to increase reusability, we define two objectives: The software components developed to derive products of a product line should be kept scalable so that they can be used for as many variants as possible. Nevertheless, the software components should be able to be reused over time in subsequent product generations. We call the latter kind of reuse the degree of usage of a software component.

Based on these prerequisites we derive a conceptual model with activities for managed evolution of automotive software product line architectures (cf. [24] and [25]) as depicted in Figure 1.2. The left part of Figure 1.2 depicts the recovery and discovery activity. This activity is performed once before the long term evolution cycle (right side of Figure 1.2) can start and repairs an initially eroded software architecture. The long term evolution cycle consists of two levels of development: The cycle on the top of Figure 1.2 constitutes the development activities for product line development, whereas the second cycle is required for product specific development. Not only

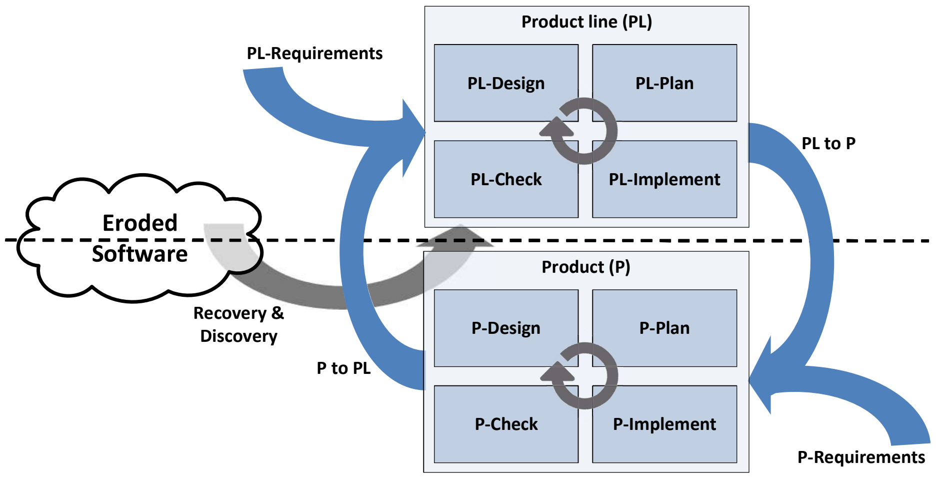


Fig. 1.2. Conceptual model for managed evolution of automotive software product line architectures

both levels of development are executed in parallel but even the activities within a cycle may be performed independently. The circular arrow within the two cycles indicates the dependencies of an activity regarding the artifacts of the previous activity. Nevertheless, individual activities may be performed in parallel, e.g., the planned implementations can be realized from activity `PL-Plan`, while a new PLA is developed in parallel (activity `PL-Design`). The large arrows between the two development levels indicate transitions requiring an external decision-making process, e.g., the decision to start a new product development or prototyping, respectively.

In the following I will briefly explain the basic activities of the conceptual model for the managed evolution of automotive software product line architectures depicted in Figure 1.2. For a more detailed definition of the activities, I refer to Section 4.1.

Software system and software component requirements from requirements engineering (`PL-Requirements`) and artifacts of the developed product from the product cycle in Figure 1.2 (`P to PL`) serve as input to the management cycle of the product line architecture (PLA). Activities `PL-Design` and `PL-Plan` aim at designing, planning and evolving product line architectures.

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`.

After sketching the conceptual model, I summarize the contribution of each of my publications:

In the paper [26] we present major challenges of automotive software engineering: Managing complexity, providing flexibility and guaranteeing dependability. This paper does not address singly one of my research objectives but instead points out challenges leading to the research objectives. We also propose basic concepts towards a managed and continuous evolution of dependable automotive software systems. The approach is demonstrated by a running example - a brake system - from the automotive domain.

Furthermore, we conduct a systematic literature study, see Section 2.2, which is currently in progress and will be published in a journal. The study is based on the conceptual model of activities for the managed evolution of automotive software product line architectures (see Figure 1.2). We are interested in the coverage of the particular aspects of the conceptual model and, thus, in the fields covered in current research and the research gaps, respectively. Furthermore, we aim to identify the methods and techniques used to implement automotive software product lines in general, and their usage scope in particular.
The following four papers address research objective one:

Automobile manufacturers cope with the erosion of their ECU software with a varying degree of success. In the paper [27] we successfully applied a methodical and structured approach of architecture restoration in the specific case of the brake servo unit (BSU). Software product lines from existing BSU variants are extracted by explicit projection of the architecture variability and decomposition of the original architecture. After initial application, this approach is capable to restore the BSU architecture recurrently.

In the paper [24] we propose an approach for repairing an eroded software consisting of a set of product architectures by applying strategies for recovery and discovery of the product line architecture. In this paper we also sketch a holistic product line management approach.

As a prerequisite for repairing an eroded software, a validated set of product line requirements is needed. These product line requirements serve as the basis for a successful manageable and evolvable PLA, too. Thus, we propose activity diagrams in conjunction with an executable semantics for requirements validation: In [28] we introduce a foundation for a framework, which enables a composition of operational semantics for activity diagrams. The composition is based on fundamental semantic constructs with options enabling the definition of domain specific variants. By our approach, a clear and intuitive operational semantics for tool development can be composed, which conforms to the informal semantics description of the UML.
The rest of my papers address research objective two:

The paper [29] presents an approach focusing on the long-term minimization of architecture erosion in the automotive domain. We introduce a description language for the specification of the logical product line architecture. Based on the description language we propose an approach for architecture compliance checking to identify architecture violations as a means to prevent architecture erosion. Our approach is demonstrated by a tool and a running example from automotive software engineering.

The work in [25] constitutes a key publication of my thesis: We propose a holistic approach for a long-term manageable and plannable software product line architecture for automotive software systems. Furthermore, we consider automotive product development and prototyping based on software product lines, and propose an approach for architecture compliance checking to avoid software erosion. The paper also includes a real world case study, a BSU software system from automotive software engineering. The case study covers a period of five years, starting with our design and implementation of a new BSU software for series production.

In [30] we deepen the approach for planning and evolving PLAs, which is a substantial part of the holistic approach for managed evolution. Our approach helps engineers to manage system complexity based on architecture design principles, techniques for architecture quality measurements and processes to iteratively evolve automotive software systems.

The work in [31] proposes an approach to extract *architectural concepts* for the design of automotive software product line architectures. 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.

Furthermore, we propose a systematic model-based, test-driven approach to design a specification on the level of modules, which is directly testable in [32]. The idea of test-driven development is to write a test case first for any new code that is written. Then the implementation is improved to pass the test case. We demonstrate our approach on a Selective Catalytic Reduction system, a real world case study from automotive software engineering.

Prof. Dr. Rausch and I organized a special track "MAAPL: Managed Adaptive Automotive Product Line Development" along with ADAPTIVE 2017 [33]. This special track focused on topics related to adaptive product line development and life-cycle management for automotive software. MAAPL contained both academic research papers as well as studies from industry introducing interesting ideas for future work.

1.5 Structure of the Thesis

This thesis is structured as follows: Chapter 2 gives an overview on the related work and presents the results of a systematic literature study on the topic of this thesis. Chapters 3 and 4 propose the approach for managed evolution of automotive software product line architectures by summarizing several of our papers. The first part of our approach - recovery and discovery of automotive software product line architectures - is introduced in Chapter 3. Next, Chapter 4 describes the holistic approach for long-term evolution of automotive software product line architectures. Finally, Chapter 5 concludes with an outlook on ongoing research.

Challenges in Developing Automotive Software Product Line Architectures: A Survey

Many challenges in the domain of automotive software product line development are mentioned in literature [34, 11]. Section 2.1 gives an overview of the corresponding literature and the related work concerning the topics of my research questions. Motivated by the huge set of challenges in this domain, there are many studies in which particular methods and techniques are proposed to support the product line driven development of software in the automotive sector (see, e.g., [34]). However, these approaches generally consider only partial aspects of development. An overview on the set of available methods and techniques would help researchers and practitioners in the field of automotive software product line engineering. Therefore, we conducted a systematic literature study and present the results in Section 2.2.

2.1 Related Work

Automotive Software Engineering

Pretschner et al. [7] provide a comprehensive overview on the state of the art in automotive software engineering. They identify research challenges in automotive software engineering, in particular the integration, evolution, maintenance, and reuse, and explored potential benefits of a seamless model-based development process as a possible solution. Furthermore, the study provides a roadmap for research in this area.

Haghighatkhaha et al. [34] recently published a systematic mapping study that classifies and analyzes the literature related to automotive software engineering. The review includes 679 articles from multiple research sub-areas, published between 1990 and 2015. They analyze research activities, topics, types, and methods and reveal research gaps in automotive software engineering. A classification of the 679 articles is listed in an excel sheet including general information for each article like abstract, title, year, authors, and research topic. This excel sheet can be accessed by an online repository[1].

Clarke et al. [35] identify essential areas of software engineering that will have a significant impact on future automotive systems and systems development. The authors introduce software product lines, global software development, service-oriented architectures, and mathematical methods applied to software engineering as future research directions for the automotive industry.

[1] See: https://www.dropbox.com/s/n7ix7h5yk9puavs/ASE_SMS_Repository_17022016.xlsx?dl=0

Grimm [36] discusses major challenges of automotive software engineering and the most important technological core competencies required to meet these challenges. Amongst other things, future work will focus on the following fields: Model-based development of distributed systems, elaboration of a product line approach for future in-vehicle software architectures, and integration of processes, methods and tools from the different areas of software, mechanical, and electrical engineering.

Gruszczynski [37] gives an overview on software engineering technologies in the automotive industry and identified future research directions. Fabbrini et al. [38] provide a picture of the achievements and the open issues in the European automotive industry and suggested future research directions.

The work in [11] presents challenges that automotive engineering faces today and discusses contributions software product line approaches could make to provide solutions for these challenges.

Software Product Line Engineering

Pohl et al. [16] propose a holistic approach on software product line engineering consisting of two key processes, domain engineering and application engineering (see Section 1.2). A core activity in software product line engineering constitutes variability management [15, 16]. Most existing approaches in variability management can be classified as feature modeling and decision modeling [39]. The main difference between both approaches is that feature modeling supports both commonality and variability modeling, whereas decision modeling focuses exclusively on variability modeling [40].

Harman et al. [41] present a survey on Search Based Software Engineering (SBSE) for SPLs and highlighted some directions for future work. They identify the most active areas in SBSE for SPLs: SPL testing, SPL feature selection, product line architecture (PLA) improvement, and SPL feature extraction.

Furthermore, several surveys on product-line testing have been conducted: Engström and Runeson [42] present a systematic mapping study on SPL testing. The main challenges identified in the paper are the large number of tests, the balance between effort for reusable components and concrete products, and handling variability. Lee et al. [43] survey the current SPL testing approaches and highlight the challenges and key research perspectives in SPL testing. They define a reference SPL testing processes as survey framework. Oster et al. [44] also address SPL testing and presented a survey on the state-of-the-art of model-based testing (MBT) approaches for SPL. They define a conceptual process model for SPL testing, which is used for a comparison of the different testing approaches. Furthermore, they highlight the challenges and open research topics in SPL testing.

Thüm et al. [45] propose a classification of SPL analyzes and survey and classify 123 existing approaches for the analysis of SPL. They also provide a research agenda to guide future research on SPL analyzes.

Chen et al. [46] discuss the findings from a systematic literature review in variability management in SPL. They present the chronological backgrounds of various approaches in variability management and identify certain gaps that need to be filled by future research: Amongst other things, they conclude, that only a few approaches address systematic process support for variability management and that there is only limited support for evolution of variability. In addition, they state the inability of most approaches to scale to large and complex product lines.

Schobbens et al. [47] present a survey on existing feature diagram variants. Based on the regarded feature diagram variants, they propose a generic formalization of the syntax and semantics of feature diagrams.

The work in [48] 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.

Measurement of Software Product Line Architecture Quality

Siegmund et al. [49] present an approach for measuring non-functional properties in software product lines. The results are used to compute optimized 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.

The work in [50] shows how automatic traceability, analyzes, 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 [51], 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 [52], 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.

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 [53] 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 [54]. In [15], 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.

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., [55, 56]. 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 [57] proposes a method that brings together two aspects of software architecture: the design of software architecture and software product lines. Deelstra et al. [58] 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.

Reference Architectures

In [59], 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 [59] 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 [60]. Furthermore, they define a reference model of reference architectures [61], and propose a methodology to design PLAs based on reference architectures [62, 63].

Software Erosion

In [22], 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 [22] 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 [22] 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 [22], none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

In [64], 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 [64], the rules are expressed as logical formulas, which can be evaluated automatically to the compliance to the PLA. These rules are extracted via Architecture Checker (ArCh) framework [65].

Van Gurp and Bosch [66] 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.

2.2 Systematic Literature Study

This section summarizes:

Christoph Knieke, Marco Kuhrmann, Andreas Rausch, Mirco Schindler, Arthur Strasser and Martin Vogel. 2017. Managed Evolution of Automotive Software Product Line Architectures: A Systematic Literature Study. Will be published in a journal.

In this section we present an in-depth literature review based on a conceptual model of artifacts and activities for the managed evolution of automotive software product line architectures. We are interested in the coverage of the particular aspects of the conceptual model and, thus, in the fields covered in current research and the research gaps, respectively. Furthermore, we aim to identify the methods and techniques used to implement automotive software product lines in general, and their usage scope in particular. We use our conceptual model (cf. Section 1.4, Figure 1.2) as a reference model to evaluate the current state-of-the-art.

The reminder of the section is organized as follows: In Section 2.2.1, we describe our research approach, and present the results of our study in Section 2.2.2. Finally, the results are discussed in Section 2.2.3.

2.2.1 Research Design

In this section, we describe our research design.

Research Method

The study at hand presents an in-depth literature study, which is grounded in [34]. Therefor, following Kitchenham et al. [67] and Petersen et al. [68, 69], we use [34] as a so-called *scoping study* that we utilize to investigate a specific topic in more detail. To investigate our research questions, we implemented the following procedure:

Step 1 In the first step, we analyzed the scoping study [34] and carried out a data cleaning based on the inclusion/exclusion criteria resulting from the research questions.
Step 2 In the second step, we removed multiple occurrences of papers in the data set following the steps described in [70].
Step 3 In the third step, we read the papers and applied the *rigor-relevance* model as proposed by Ivarsson and Gorschek [71].
Step 4 The forth step comprised a quality assessment following the approach described in [72].
Step 5 In the fifth step, we prepared the in-depth review. In this regard, we used the conceptual model for software product lines from Section 1.4, which we used to further classify the papers in the data set.
Step 6 Finally, we used the aforementioned conceptual model and the categorized papers to conduct the in-depth review to answer the research questions.

The research approach above was implemented in a team of researchers with well-defined task distribution to improve the validity of the findings.

Research Questions

With the study at hand, we aim to understand the current state-of-the-art in holistic approaches for a managed evolution of software product line architecture in automotive software engineering. Specifically, we define our working hypothesis as follows: *There is no holistic approach for a managed evolution of automotive software product lines.*

Table 2.1. Research questions addressed with the study at hand.

	Research question and rationale
RQ_1	*What is the current state-of-the-art in holistic approaches for managed evolution of automotive software product lines?* We aim at collecting information about such holistic approaches and use our conceptual model (cf. Section 1.4) as a reference model to evaluate the current state-of-the-art. Specifically, we are interested into the coverage of the particular aspects of the conceptual model and, thus, the fields covered in current research and research gaps, respectively.
RQ_2	*What particular methods and techniques are used to implement a managed evolution of automotive software product lines?* We aim to identify the methods and techniques used to implement automotive software product lines in general, and their usage scope in particular. For this, we analyze the available literature and categorize and evaluate the contributions found according to a given schema (Section 1.4).

Data Collection Procedures

The data collection is based on a previously conducted study [34], which we use as a scoping study. In the following, we provide a brief summary of the scoping study's contribution, before we describe the selection process of papers relevant to the study at hand. Finally, we describe how we selected the final set of papers for analysis.

As basic data source, we use the scoping study [34] and the complementing published dataset. The scoping study has collected papers on Automotive Software Engineering, which were categorized into the seven research areas listed in Table 2.2.

In total, the scoping study [34] includes 679 papers, which were categorized into seven research areas and 14 specific research topics. This dataset served as input for the data collection, which is explained in detail in the following sections.

As introduced the scope of the study at hand is the field of software product lines. For this, we used the scoping study and applied a multi-staged selection procedure for studies of interest:

1. Analyze the scoping study's data and identify all papers on software product lines (incl. synonyms):
   a) Selection of papers based on title and abstract
   b) Selection based on keywords[2] (incl. variants, upper-/lower case, etc.): SPL, family, reference architecture, variability, variant model, variability management, feature model, feature tree, feature-oriented, derivate

[2] While testing the selection criteria, we decided to exclude the keywords *reusability, reuse, derive,* and *feature*, since they generated too many false positives.

Table 2.2. Automotive Software Engineering research areas according to ISO/IEC 12207 PRM as found in [34].

Research area and topics	**Total number of studies per area**[a]
Agreement Processes (AGR)	5
• Agreement Support Group	
Organizational Project-Enabling Processes (ORG)	48
• Organizational Project-Enabling Support Group	
Project Processes (PRO)	14
• Project Support Group	
Technical and Software Implementation Processes[b] (ENG/DEV)	439
• System/Software Architecture and Design (131 studies). • System/Software Qualification Testing (127 studies) • Software Implementation (62 studies). • System/Software Integration (44 studies) • System/Software Requirement Engineering (35 studies). • Software Construction (22 studies). • Software Maintenance (18 studies).	
Software Support Processes (SUP)	122
• Software Verification and Validation (71 studies). • Software Quality Assurance and Review (48 studies). • Software Documentation and Configuration Management (3 studies).	
Software Reuse Processes (REU)	72
• Software Reuse	

[a] Multiple assignments possible
[b] Summarized

   c) *Result:* This stage resulted into 87 paper (candidates); most of the papers selected are in the REU group (Table 2.2)
2. Rating of the study candidates
   a) Application of the rigor-relevance model [71]
   b) Definition of a Threshold and selection of papers
   c) Final data cleaning and preparation of the in-depth analysis
3. In-depth analysis

As mentioned before, in the final paper selection, we analyzed the 87 selected candidate studies and, in a first step, applied the rigor-relevance model as proposed by Invarsson and Gorschek [71]. This model grades papers on the two parameters *rigor* (FROM-TO) and *relevance* (FROM-TO). To select the studies of interest, we defined the threshold $rigor + relevance \geq 2$ to ensure that:

1. All high-quality papers (i.e., high-scored papers) are in the result set.
2. Papers that have a high relevance score but a poor rigor score are included.
3. Medium-scored but balanced papers are included.

Applying the rigor relevance model to the candidate studies, we selected 40 studies. These studies were (again) checked with a particular focus on multiple occurrences. In order to finally clean the dataset, we decided to prefer journal papers to conference papers, as we assume follow-up special issue papers to have a higher maturity/quality. Eventually, we selected *35* primary studies for inclusion into the data analysis. The final evaluation and classification of the different studies is summarized in Table 2.3. A quick overview of the studies selected can be found in Table 2.4.

Table 2.3. Evaluation and classification of the primary studies selected for analysis.

Id	Ref	RTF[a]	CTF[b]	Rigor	Relevance
P8	[73]	Evaluation	Model	1.0	2.0
P22	[74]	Solution Proposal	Framework/Method/Technique	1.5	2.0
P43	[75]	Evaluation	Lessons Learned	3.0	3.0
P71	[76]	Evaluation	Framework/Method/Technique	1.5	2.0
P72	[77]	Evaluation	Lessons Learned	3.0	3.0
P79	[78]	Evaluation	Framework/Method/Technique	1.0	2.0
P86	[79]	Evaluation	Lessons Learned	2.0	4.0
P90	[80]	Evaluation	Model	2.0	3.0
P94	[81]	Evaluation	Framework/Method/Technique	1.0	2.0
P100	[82]	Validation	Framework/Method/Technique	2.5	1.0
P129	[83]	Evaluation	Model	2.0	4.0
P173	[84]	Evaluation	Framework/Method/Technique	1.0	2.0
P215	[85]	Evaluation	Lessons Learned	2.5	4.0
P218	[86]	Experience Paper	Lessons Learned	0.5	2.0
P220	[87]	Experience Paper	Lessons Learned	0.5	2.0
P221	[88]	Experience Paper	Lessons Learned	0.5	2.0
P223	[89]	Experience Paper	Framework/Method/Technique	1.0	2.0
P279	[90]	Evaluation	Framework/Method/Technique	1.0	3.0
P281	[91]	Validation	Guideline	1.0	2.0
P285	[92]	Evaluation	Framework/Method/Technique	1.5	4.0
P289	[27]	Evaluation	Framework/Method/Technique	1.0	2.0
P300	[93]	Evaluation	Lessons Learned	1.0	2.0
P310	[94]	Evaluation	Framework/Method/Technique	1.0	2.0
P332	[95]	Evaluation	Framework/Method/Technique	1.0	1.0
P343	[96]	Evaluation	Framework/Method/Technique	1.0	2.0
P365	[97]	Evaluation	Framework/Method/Technique	1.0	1.0
P377	[98]	Solution Proposal	Framework/Method/Technique	1.0	1.0
P404	[99]	Evaluation	Lessons Learned	0.5	2.0
P468	[100]	Experience Paper	Lessons Learned	1.0	3.0
P493	[101]	Experience Paper	Framework/Method/Technique	2.0	3.0
P503	[102]	Experience Paper	Lessons Learned	0.0	2.0
P580	[103]	Solution Proposal	Framework/Method/Technique	1.0	3.0
P588	[104]	Evaluation	Framework/Method/Technique	1.0	3.0
P589	[105]	Evaluation	Framework/Method/Technique	1.0	2.0
P660	[106]	Evaluation	Framework/Method/Technique	1.0	2.0

[a]Research type facet, according to [107]
[b]Contribution type facet, according to [68, 69]

Data Extraction and Dataset Quality Assessment

The data extraction and the quality assessment was performed in a combined manner. For this, we developed a data sheet that was filled paper-wise. Table 2.5 shows the data extraction sheet. The actual outcome of the quality assessment of the papers included in the study can be taken from Table 2.6, respectively.

Table 2.4. Overview of the primary studies selected for in-depth data analysis.

Id	Ref	Title
P8	[73]	A component-based process for developing automotive ECU software
P22	[74]	A model-based approach to innovation management of automotive control systems
P43	[75]	A Survey on the Benefits and Drawbacks of AUTOSAR
P71	[76]	An unadjusted size measurement of embedded software system families and its validation
P72	[77]	Analysing defect inflow distribution of automotive software projects
P79	[78]	AORE (aspect-oriented requirements engineering) methodology for automotive software product lines
P86	[79]	Architecting automotive product lines: Industrial practice
P90	[80]	Architecture for embedded open software ecosystems
P94	[81]	Assessing merge potential of existing engine control systems into a product line
P100	[82]	Automated diagnosis of feature model configurations
P129	[83]	Automotive system development using reference architectures
P173	[84]	Defining a strategy to introduce a software product line using existing embedded systems
P215	[85]	Evolutionary architecting of embedded automotive product lines: An industrial case study
P218	[86]	Experience of Introducing Reference Architectures in the Development of Automotive Electronic Systems
P220	[87]	Experience with variability management in requirement specifications
P221	[88]	Experiences from a large scale software product line merger in the automotive domain
P223	[89]	Experiences of applying model-based analysis to support the development of automotive software product lines
P279	[90]	IVaM: Implicit variant modeling and management for automotive embedded systems
P281	[91]	Lightweight introduction of EAST-ADL2 in an automotive software product line
P285	[92]	Managing complexity and variability of a model-based embedded software product line
P289	[27]	Mastering Erosion of Software Architecture in Automotive Software Product Lines
P300	[93]	Model transformation for high-integrity software development in derivative vehicle control system design
P310	[94]	Model-based pairwise testing for feature interaction coverage in software product line engineering
P332	[95]	On hardware variability and the relation to software variability
P343	[96]	Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems
P365	[97]	Relating requirement and design variabilities
P377	[98]	Reuse of Software in Distributed Embedded Automotive Systems
P404	[99]	Software behavior description of real-time embedded systems in component based software development
P468	[100]	Towards integrated variant management in global software engineering: An experience report
P493	[101]	Variation management for software product lines with cumulative coverage of feature interactions
P503	[102]	Why does it take that long? Establishing Product Lines in the Automotive Domain
P580	[103]	A process to support a systematic change impact analysis of variability and safety in automotive functions
P588	[104]	A model-based approach to support the automatic safety analysis of multiple product line products
P589	[105]	Supporting the automated generation of modular product line safety cases
P660	[106]	Evaluating flexibility in embedded automotive product lines using real options

Analysis Procedures

The analysis procedures applied to the final dataset of 35 papers also followed a multi-staged approach. In our analysis procedure, we applied a number standard analyses on the data already provided by the original scoping study [34]. In particular, we used the provided data for analyzing the *research type facets* [107] and the *contribution type facets* [68, 69]. Furthermore, we applied the rigor-relevance model [71] for implementing study selection and further data analyses.

Grounded in these basic measures, we used the conceptual model for a managed evolution from Section 1.4 as a classification schema. Five researchers classified the papers according to the conceptual model from Section 1.4, which we used as a classification schema. For each element in the classification schema, the individual researcher had to state whether or not a study makes a major contribution to a specific element, e.g., *product line design* or *product implementation.* The results were integrated and checked for agreement using Fleiss' κ [108]; the overall agreement in

Table 2.5. Data extraction and study quality assessment sheet used in this study.

Category	**Questions**
Data Extraction	The data extraction was performed using the following key questions: 1. Which methods and or techniques are used? • Are the same methods used for domain and application engineering? • Are the methods/techniques used evaluated (see quality assessment)? 2. Is there a holistic approach? • Are the different activities (Section 1.4) fully covered? • Are there gaps? • What are the consequences for a holistic approach?
Quality Assessment	The quality assessment was carried out following the list of questions from Kitchenham and Charters [72, p. 28] by asking the following questions: • 1. How credible are the findings? • 1.1. If credible, are they important? • 3. How well does the evaluation address its original aims and purpose? • 4. How well is the scope for drawing wider inference explained? • 5. How clear is the basis of evaluative appraisal? • 6. How defensible is the research design? • 7. How well defined are the sample design/target selection of cases/documents? • 8. How well is the eventual sample composition and coverage described? • 9. How well was data collection carried out? • 10. How well has the approach to, and formulation of, analysis been conveyed? • 11. How well are the contexts and data sources retained and portrayed? • 12. How well has diversity of perspective and context been explored? • 13. How well have detail, depth, and complexity (i.e. richness) of the data been conveyed? • 14. How clear are the links between data, interpretation and conclusions – i.e. how well can the route to any conclusions be seen? • 15. How clear and coherent is the reporting? • 16. How clear are the assumptions / theoretical perspectives/values that have shaped the form and output of the evaluation? • 18. How adequately has the research process been documented? Each question was rated on a 5-point scale with: *criterion is* 1=not fulfilled at all, 2=partially fulfilled, 3=basically fulfilled, 4=fulfilled to a large extent, and 5=completely fulfilled and very well implemented/documented. Please note that the numbers for the individual questions are taken from [72, p. 28]. In Table 2.6, we refer to these numbers to present the quality assessment for the particular studies.

Table 2.6. Quality assessment of the papers included in this study.

Paper	**Ref**	**1**	**1.1**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**18**
P8	[73]	3	3	3	3	4	3	4	4	3	3	3	2	2	3	4	3	3
P22	[74]	4	4	5	3	3	3	2	3	2	3	3	3	3	4	4	3	4
P43	[75]	5	5	5	4	5	5	4	4	5	5	5	3	4	4	4	4	5
P71	[76]	5	4	4	3	4	4	3	3	3	3	3	3	3	4	4	3	4
P72	[77]	5	4	4	4	4	4	4	4	4	4	4	5	4	4	5	4	4
P79	[78]	3	3	3	3	3	2	3	3	2	3	3	2	2	3	3	3	3
P86	[79]	4	4	4	4	4	4	3	3	4	4	3	3	3	4	4	4	4
P90	[80]	5	4	4	3	3	4	3	3	4	4	4	4	3	4	4	4	4
P94	[81]	4	3	4	3	3	3	3	3	2	2	3	2	2	3	3	3	3
P100	[82]	4	3	4	4	4	5	4	4	5	4	4	4	4	5	5	4	4
P129	[83]	4	4	5	4	4	5	4	4	4	4	4	4	3	4	4	3	4
P173	[84]	4	3	4	3	3	3	3	2	2	2	3	3	2	3	3	3	3
P215	[85]	4	4	5	4	5	4	3	3	4	4	3	4	3	4	4	4	4
P218	[86]	4	4	4	3	2	3	2	2	3	2	3	4	2	3	3	2	3
P220	[87]	4	4	4	3	3	2	2	2	3	2	3	3	2	3	3	3	2
P221	[88]	4	4	3	3	3	3	2	3	2	3	3	2	3	3	3	3	3
P223	[89]	4	4	3	2	3	3	2	2	2	2	3	3	2	3	3	3	3
P279	[90]	4	3	4	3	2	3	2	2	2	2	2	3	2	3	3	2	3
P281	[91]	3	3	3	2	2	3	2	2	2	2	3	3	2	3	3	2	3
P285	[92]	4	4	4	3	3	3	3	3	3	3	3	2	2	3	4	3	3
P289	[27]	4	4	4	3	2	3	2	3	2	2	3	3	2	3	3	3	3
P300	[93]	4	3	4	3	3	3	4	3	3	3	3	2	2	3	3	3	3
P310	[94]	4	4	4	4	3	3	3	3	3	3	2	2	2	3	4	4	3
P332	[95]	4	4	4	3	4	3	3	3	2	3	3	3	2	3	4	4	3
P343	[96]	4	4	4	3	3	3	3	3	3	3	3	3	3	3	4	3	4
P365	[97]	4	4	3	3	2	3	2	3	3	3	2	3	3	3	3	3	3
P377	[98]	4	4	3	2	2	3	3	2	2	3	3	3	2	3	4	3	3
P404	[99]	4	4	4	3	3	3	3	3	2	3	2	2	2	3	4	3	3
P468	[100]	4	4	4	4	4	4	3	3	4	3	3	3	2	3	4	3	3
P493	[101]	4	4	4	3	4	4	3	3	3	4	3	3	4	4	4	4	4
P503	[102]	3	2	3	2	2	2	1	1	2	2	2	2	2	3	3	3	2
P580	[103]	4	4	3	4	4	3	3	4	3	3	3	3	3	3	4	4	4
P588	[104]	4	4	3	3	3	3	3	3	3	3	3	2	3	3	3	3	3
P589	[105]	4	4	3	3	3	3	3	3	3	3	3	2	2	3	3	3	3
P660	[106]	3	3	3	2	3	3	3	3	2	3	2	2	2	3	3	2	3

the studies' classification was 0.72 (substantial agreement). This classification resulted into three categories:

Category 1 This category includes those papers for which all five researchers agreed that a study has a major contribution in a specific category. Finally, four papers were assigned to this category.

Category 2 This category includes those papers for which three or four researchers agreed that a study has a major contribution in a specific category. Finally, 22 papers were assigned to this category.

Category 3 This category includes those papers for which one or two researchers only agreed that a study has a major contribution in a specific category. Finally, nine papers were assigned to this category.

The categories above are used in further in-depth analyses to, notably, evaluate the specific contributions and the maturity of the contributions of the individual studies. Finally, we used the classification schema from Section 1.4 and the researchers' rating to generate a mapping table, which guides the in-depth content analysis.

Validity Procedures

As we reused an already published dataset resulting from an independently conducted scoping study [34], we implemented several measures to improve the validity of our findings. First and foremost, we relied on researcher triangulation, i.e., we always ensured that one researcher from the team was not involved in a task and performed the quality assurance of that very task, e.g., in the rating of the contributions of the study, five researchers evaluated the studies and a sixth researcher checked the evaluations and computed the agreement levels. Furthermore, we shuffled the teams, e.g., data collection, quality assessment, and data analysis was performed by different teams of two to three researchers.

2.2.2 Study Results

This section summarizes the results of our analysis. First, we provide general demographic information about the dataset, before we answer the individual research questions.

Result Overview

After the study selection, 35 papers remained in the result set. Figure 2.1 illustrates the publication frequency including the 3-year trend (red line) and the 5-year trend (black line). The general publication frequency shows a growing interest in the research on automotive software product lines starting in 2011.

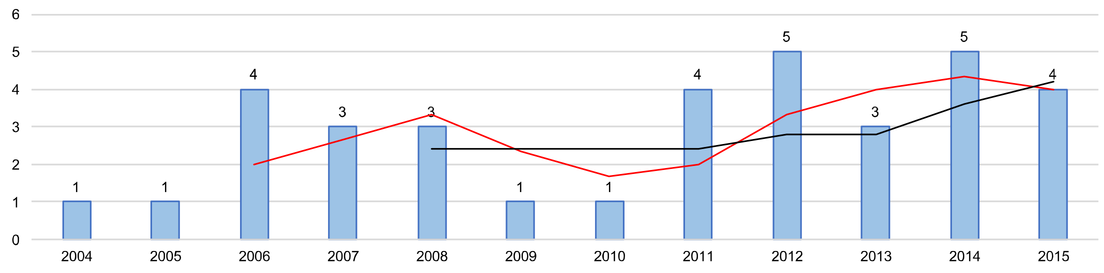


Fig. 2.1. Overview of the publication frequency including publication trend.

Based on the initial classification from the scoping study [34], we evaluated the remaining 35 papers for their research- and contribution type classification, which is illustrated in Figure 2.2. This classification shows the majority of the papers contributing frameworks, methods and techniques, and that most studies are classified as *evaluation research*. It has to be noted that the papers from our result set have been filtered using the rigor-relevance model (see also Table 2.3). Hence, due to the selection procedure, we expected mostly papers of type evaluation research. Yet, the result set contains 23 papers of type evaluation research, which we consider an indication towards practically relevant contributions. Furthermore, seven more papers from the result set have been classified as *experience paper*, i.e., practical experience regarding automotive software product lines have been reported.

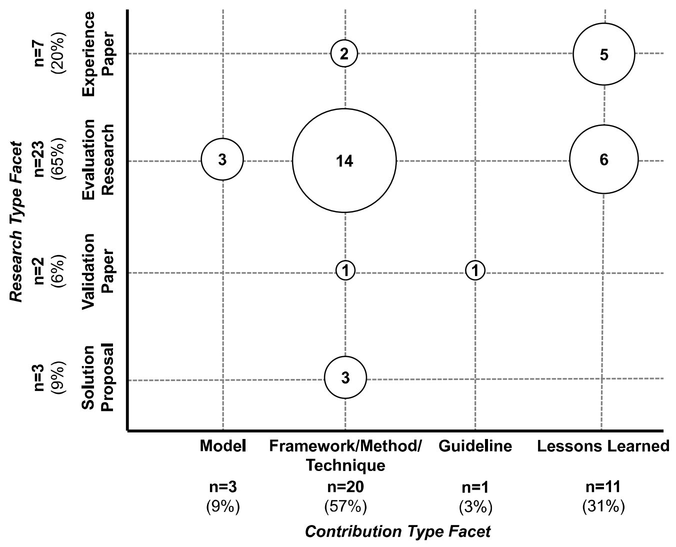


Fig. 2.2. Classification of the result set according to research typ facets and contribution type facets.

In a nutshell, we argue that our paper selection strategy resulted in a dataset that comprises those studies reporting practical relevant knowledge about automotive software product lines, which we investigate in more detail in the following sections.

State of the Art in Holistic Approaches for Managed Evolution of Automotive Software Product Lines

In this section, we aim to answer research question RQ_1 (Table 2.1). As described in Section 2.2.1, five researchers classified the papers according to the conceptual model from Section 1.4. The classification was used to generate a mapping table for the in-depth content analysis. Based on this mapping table, Figure 2.3 shows the assignment of the papers to the conceptual model from Section 1.4. In Figure 2.3 a paper is annotated to an element if at least two researchers state that the paper makes a major contribution to that specific element. A paper can be assigned to more than one element if it contributes to several activities of the conceptual model.

Figure 2.3 shows the assignments of the selected primary studies to the activities of our classification schema. According to our schema, none of the analyzed studies presents a holistic approach for managed evolution of automotive software product lines. Yet, some studies provide a certain coverage. For instance, study P377 [98] has been assigned to nine categories thus covering three-quarters of the activities of our schema. It has also to be mentioned that this study only received a score of 1.0 in the rigor-relevance evaluation (Table 2.3), i.e., it has barely received the minimum score required for inclusion in the result set. The two primary studies P22 [74] and P129 [83] have been assigned four times each, and studies P218 [86], P365 [97], and P493 [101] have been assigned thrice. The remaining studies have been assigned once or twice only. In the subsequent section we will regard all studies proposing an “overall approach/process” in detail.

To facilitate the evaluation we count the assignments of studies to activities of the classification schema (Figure 2.3) and display the percentages of counts in Table 2.7. In total, there are 70 assignments to the 13 activities of the classification schema, i.e., a study has been assigned on average to two activities. Two studies (P343 [96], P589 [105]) have been assigned to category

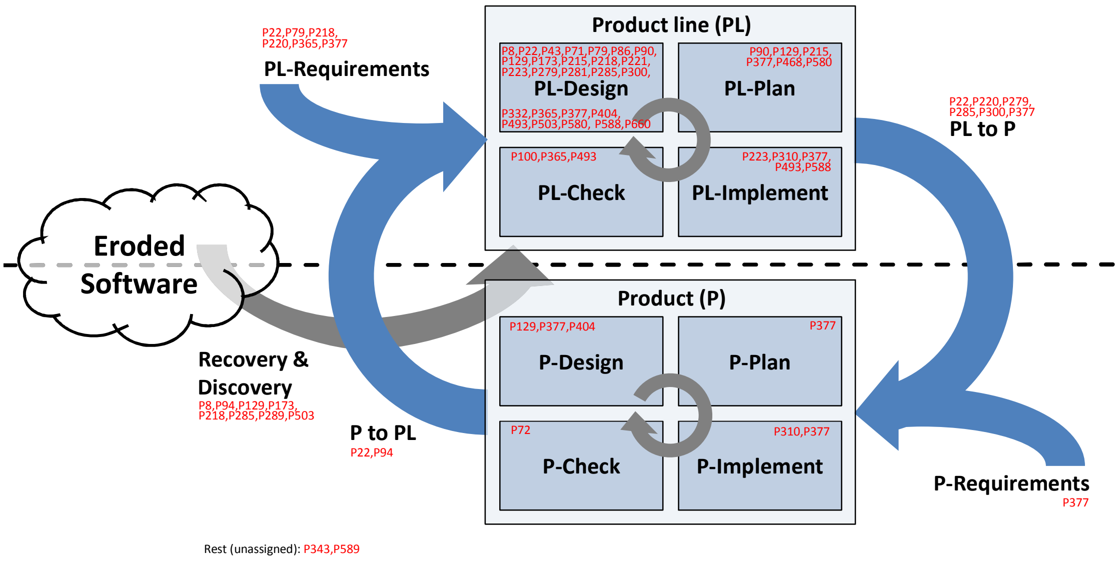

Product line (PL)
P8,P22,P43,P71,P79,P86,P90, P129,P173,P215,P218,P221, P223,P279,P281,P285,P300, P332,P365,P377,P404, P493,P503,P580, P588,P660
PL-Design
P90,P129,P215, P377,P468,P580
PL-Plan
P100,P365,P493
PL-Check
P223,P310,P377, P493,P588
PL-Implement
P22,P79,P218, P220,P365,P377
PL-Requirements
P22,P220,P279, P285,P300,P377
PL to P
Eroded Software
Recovery & Discovery
P8,P94,P129,P173, P218,P285,P289,P503
P to PL
P22,P94
Product (P)
P129,P377,P404
P-Design
P377
P-Plan
P72
P-Check
P310,P377
P-Implement
P-Requirements
P377
Rest (unassigned): P343,P589

Fig. 2.3. Assignments of the finally selected primary studies to the classification schema (Section 1.4) to prepare the in-depth content analysis.

“Rest (unassigned)”. Here, none of the 13 activities in our classification schema has received at least two assignments for the corresponding study.

Figure 2.3 and Table 2.7 reveal that the contribution of the studies concentrate on `PL-Design` (37%), followed by `Recovery & Discovery` (11%), `PL-Requirements` (9%), `PL-Plan` (9%), and `PL to P` (9%). Based on the accumulation of studies on the initial activities of the schema we hypothesize that the studies in question are primarily concerned with the development of new SPL rather than the long-term evolution of the product line. For example, architecture conformance checking related activities required to prevent architecture erosion in the long term is underrepresented in the studies: Only four studies (P72 [77], P100 [82], P365 [97], P493 [101]) can be found that our researchers have mapped to this appropriate fields.

Table 2.7. Number of assignments to the classification schema in Figure 2.3

	Product Line (PL)	**Product (P)**
Design	26 (37%)	3 (4%)
Plan	6 (9%)	1 (1%)
Implement	5 (7%)	2 (3%)
Check	3 (4%)	1 (1%)
Requirements	6 (9%)	1 (1%)
PL to P	6 (9%)	
P to PL	2 (3%)	
Recovery & Discovery	8 (11%)	
Rest (unassigned)	2	

Remarkably few studies address the product specific development cycle (application engineering) and only study P72 [77] exclusively addresses the product specific development cycle. One reason for this could be that no specific methods and techniques are required for the product development cycle within a product line based development approach.

Methods and Techniques to Implement Automotive Software Product Lines

In this section, we aim to answer research question RQ_2 (Table 2.1). Specifically, we identify the methods and techniques used to implement automotive software product lines in general, and their application scope in particular. Table 2.8 provides a categorization, which is built based on the in-depth content analysis (see Table 2.5). Furthermore, the table provides an overview of the different studies and how their major contributions are applied, i.e., in the scope of the product line as such and in the context of particular products derived from that product line. In the following, we first provide an overview before we provide the details structured following the categories listed in Table 2.8.

From the content analysis of the 35 selected primary studies we derived 12 categories—each representing a cluster of methods and/or techniques. As already discussed, most of the papers propose one method or technique and provide—if at all—a scoped evaluation only. That is, Table 2.8 lists the topics of interest and, at the same time, provides some information about topics of interest in automotive software product line research, however, based on the result set, we can only derive a "fragmented" picture as several aspects are studies, yet in a fairly isolated manner. Nonetheless, the result set shows that some topics are studied from different perspectives, which indicates the variety of the subject field.

In the following, we discuss the different method/technique clusters individually and in more detail.

Table 2.8. Methods and or techniques of the papers included in this study.

No.	Methods and/or techniques	Papers	PL[a]	P[b]
1	Architecture evolution process	[79, 83, 85]	x	x
2	Cost/effort estimation	[76, 81, 84, 106]	x	
3	Safety analysis	[77, 103, 104, 105]	x	
4	Description languages	[99]	x	x
5	Architecture reengineering	[27]	x	
6	Model transformation	[82, 89, 91, 92, 93]	x	
7	Model-based requirements engineering	[74, 78]	x	x
8	Overall approach/process	[92, 98, 102, 83, 86]	x	x
9	Reference architectures	[75, 80, 83, 86]	x	x
10	SPL merging	[81, 84, 88]	x	
11	Testing / Verification	[94, 96]	x	x
12	Variability management	[73, 87, 90, 95, 97, 100, 101, 103, 104, 105]	x	x

[a]Activities related to product line development (domain engineering) including Recovery & Discovery
[b]Activities related to product development (application engineering)

Architecture Evolution Process

This section discusses the studies P86 [79], P129 [83] and P215 [85], which were found addressing the architecture evolution process category—notably the maintenance and the long-term evolution of software product line architectures.

Gustavsson and Eklund [79] discuss the architects' work approach concerning the maintenance in the context of an evolving/changing *product line architecture* (PLA). Authors conducted a series of expert interviews with architects from Scania and Volvo Car. They found that the process for managing changes of the PLA is very similar and that the found processes from both companies can be mapped to a generic process, which consists of the five steps *need*, *impact analysis*, *solution*, *decision*, and *validation*. All of these findings are based on conclusions concerning the interviews and are not verified by a further case study.

The work of Lind and Heldal [83] is primarily focused on *reference architectures*, but also addresses the evolution of architectures. They distinguish between revolutionary and evolutionary architecture processes, which are both discussed in [83]. In particular, authors argue that reference architectures have to be continuously maintained, notably, it is necessary to evaluate reference architectures continuously to identify bottlenecks, which is key to initiate refactoring and to support the architecture's evolution. Hence, Lind and Heldal present an empirically-grounded *reference architecture development process*, which contains activities that are primarily performed for evolutionary architecting, e.g., activities "Synthesize, Evaluate, and Verify & Validate Architecture". Such activities describe the architecture design, the measurement of architecture quality, and the validation against the requirements in the context of an evolving architecture.

Axelsson [85] discusses *revolutionary architecture process* (RAP) and *evolutionary architecture processes* (EAP). The specifically author focuses on the interplay of both processes and on how the EAP is performed in practice. In an empirical study, reasons for changes in an architecture are discussed, complemented by a discussion of affected attributes, technical aspects involved, and decisions made. The study shows that RAP and EAP differ significantly. Also, Axelsson states that most literature mainly describes RAP, whereas EAP lack studies and evidence.

In summary, the primary studies assigned to this category use similar activities for analyzing/describing the evolution of (automotive) software product line architectures. Key is the process-support of the architecture processes, notably for architecture development and maintenance/evolution. A cross-company analysis conducted by Gustavsson and Eklund [79] did reveal similarities, which underpin the usefulness of a generalized/holistic concept.

Cost-/Effort Estimation

This section discusses the studies P71 [76], P94 [81], P173 [84], and P660 [106], which were found addressing the cost-/effort estimation category. A number of studies consider the cost-/effort estimation a key activity, especially in the course of planning a product line development endeavor.

Kiebusch et al. [76] propose metrics to measure the size of an automotive software product line. They argue that neither of the existing methods adequately measures the (unadjusted) size nor estimates the cost of process-oriented automotive software product lines. For this, authors propose the *Process-Family-Points* (PFP) analysis method to allow for size measurement and effort estimation.

Yoshimura et al. [81, 84] address effort estimation methods with two studies: The approach proposed in [81] describes a software clone analysis approach, and the approach discussed in [84] proposes an estimation process based on the return on investment (ROI). Both approaches are integrated into a process for merge potential assessment of existing variants, i.e., how to (economically) reintegrate variants back into the product line. In [84], authors discuss an application of this approach and present lessons learned and open issues. As lessons learned they discuss, e.g., that software cloning may not be a good way to realize product line engineering, and that ROI predictions can strongly motivate the management to invest in product line engineering, and that architecture-centric clone analysis is a useful and practical approach to assess the merge potential of the existing systems. Open issues are, e.g., clone visualization, clone refactoring, and clone error reduction.

Gustavsson and Axelsson [106] provide a method of evaluating system designs with the purpose of enabling practitioners to systematically think about the future development of a system. For this, they use the *Real Options Theory* [109, 110] that adds the possibility to put an economic value on the system adaptability attribute and, thus, motivates architects to also anticipate the actual value of future developments of an architecture.

In summary, the studies [76, 84, 106] use different methods to perform cost- and effort estimation actives in the course of planning software product line development. Furthermore, the studies provide process models for practically applying these methods, and the studies present evaluations of the respective approaches.

Safety Analysis

This section discusses the studies P72 [77], P580 [103], P588 [104], and P589 [105], which were found addressing the safety analysis category.

Two studies address safety analysis for the development of safety-critical automotive software product lines. Rana et al. [77] address the problem of selecting the appropriate *Software Reliability Growth Model* (SRGM) of more than 100 currently existing SRGMs. Growth models are used for evaluating the maturity or release readiness of a software before its release and, respectively, for an optimal allocation of the test resources required. Rana et al. use a statistical model to identify the distribution of defects, which helps selecting an appropriate growth model. A case study conducted at Volvo Car Group is utilized to evaluate the proposed approach. They evaluate six standard distributions on defect inflow data from four large software projects and show that beta distribution provides the best fit to the defect inflow data. Second, Käßmeyer et al. [103] present an improved safety engineering approach for software product line development. Their approach provides an integrated change impact analysis by combining their approach with variability management. They apply their approach to an industrial example, a small part of an Advanced Driver Assistant System (ADAS), to illustrate the benefits. As demonstrated by the example, changes are propagated in one model for both variant management and safety engineering.

De Oliveira et al. [104] propose an approach to support the generation of fault trees and FMEA analyses for products derived from a software product line. Their approach aims at reducing the effort required for performing safety analyses for the products. The study proposes a process model for model-based safety analysis, which starts with a product line hazard analysis, followed by a process step "Augmentation of PLA with failure logic" that describes how product line architecture design elements, i.e., product line components, can fail and how they contribute to the occurrence of hazards. Further steps include the definition of software product line configuration knowledge, product derivation, and safety assessment. In [105], de Oliveira et al. propose an approach to support the automated construction of modular product line safety cases. The approach uses different techniques like architecture failure modeling, functional failure modeling, and component failure modeling. Note, that the three studies [103, 104, 105] are also assigned to the category variability management for which we provide a further discussion.

Description Languages

This section discusses the study P404 [99], which was categorized in the description languages category. For the development of a software product line, a suitable description technique is key for supporting the specification of variability. Kim et al. [99] propose a concept to support the functional view (software behavior) in the component-based software development (CBSD). The approach introduces "signal flows" and "mode-dependent signal flows" for the specification of a component. In this regard, important information is provided at the component level to support the understanding of the software's behavior and to also understand dependencies among software components when reusing and adapting software components. The approach is proposed in the context of product line development in the automotive sector. However, the study does not provide extensive details concerning the degree of appropriateness of the approach for product line-driven development. The approach rather addresses a more general problem of software development in the automotive sector.

Architecture Reengineering

This section discusses the study P289 [27]), which addresses the architecture reengineering. Architecture erosion has become a major challenge in automotive software engineering, which results in a considerable effort for maintenance and software system evolution. Furthermore, architecture erosion is a major problem affecting software reuse [24]. In this context, Strasser et al. [27] propose an approach for reengineering an eroded software product line architecture. First, all relevant variation points and the associated functional requirements of a component are identified. A variability analysis is performed by the *Product Line UML-based Software Engineering* (PLUS; [54]) approach, which is also used to describe appropriate variability models. Based on the analysis results of the product line extraction process, architecture components are identified and designed in the next step. This approach can also be assigned to the recovery & discovery activity.

Model Transformation

This section discusses the studies P100 [82], P223 [89], P281 [91], P285 [92], and P300 [93], which were assigned to the model transformation category. Several studies utilize model transformation techniques to transform artifacts of software product line development into different models, such that techniques, e.g., for model analysis can be applied.

For instance, White et al. [82] propose an approach to debugging feature model configurations and automating configuration evolution, called CURE (Configuration Understanding and REmedy). Configurations and feature models can be transformed into constraint satisfaction problems (CSP) to automatically diagnose errors and repair invalid feature selections.

Merschen et al. [89] present a prototypical framework for the analysis of embedded software product lines. They analyze artifacts by transforming them into models, which are used in an analysis process based on model transformation languages. The automated preprocessing is implemented as model transformations in ATLAS Transformation Language (ATL) and Epsilon Transformation Language (ETL).

Leitner et al. [91] introduce EAST-ADL2 in an automotive software product line including a transformation from AUTOSAR to EAST-ADL2. Basic variability information can be automatically extracted during the transformation step. Different mapping strategies are analyzed to generate a correct model and to reduce losses in the transformation process. Furthermore, they describe the implementation of the transformation process.

Polzer et al. [92] present a framework for model-based product lines of embedded systems. The framework supports the (semi-)automated extraction of models from existing requirement-, test-, and implementation artifacts.

Finally, Wang [93] presents a study on the application of model transformation techniques on the development of automotive software product lines. The paper aims at understanding the state-of-the-art techniques and to identify model transformation challenges in product-line-based automotive software—notably to help developers choosing the model transformation technique appropriate for the respective situation. Wang distinguishes between model transformation at the same abstraction level and model transformation across different abstraction levels. The transformations are implemented by using the tool GReAT (Graph Rewrite And Transformation). Wang presents the results of a case study with a simplified enhanced cruise control system (eCCS). As lessons learned, he states that model transformations with well-design transformation rules yield consistent implementations across vehicle variations and can thus reduce the efforts to create and maintain the design variations for different vehicles. In addition, Wang reveals that current transformation features partially meet the needs of derivative design, and still require improvement.

In summary, the studies [82, 89, 91, 92, 93] show that model transformation techniques play a key role in the development of automotive software product lines. By means of model transformations, tools and techniques can be used in the different process steps. The effort to create and maintain design variations, e.g., for different vehicles, can be reduced. Well-design transformation rules are crucial for the effectiveness of a model transformation.

Model-based Requirements Engineering

This section discusses the studies P22 [74] and P79 [78], which were assigned to the model-based requirements engineering category. Requirements engineering requires appropriate support by methods and techniques to capture and manage the requirements for a software product line to maintain the high degree of variability.

For this, Gleirscher et al. [74] present an approach for integrating innovation management with requirements and technology management. Requirements-based innovations are usually motivated by newly elicited requirements or needs originating from market research, whereas technology-based innovations are motivated by new or emerging technologies (e.g., specific platform components and platform services). To identify innovations, they use models of feature hierarchies, platform service hierarchies, and a platform component models. Furthermore, to accomplish innovations they define activities for requirements-based innovations and technology-based innovations.

Aoyama and Yoshino [78] present an aspect-oriented approach for the requirements engineering in software product line development. Automotive systems often deal with a wide spectrum of interwoven functional and nonfunctional requirements. They apply a multi-dimensional aspect-oriented modeling and analysis to generate multiple software product lines for automotive systems. The approach separates intersecting non-functional requirements (NFR) into primitive concerns (aspects) and introduces quantitative metrics of each NFR/aspect.

Overall Approach/Process

This section discusses the studies P285 [92], P377 [98], P503 [102], P129 [83], and P218 [86], which were assigned to the overall approaches/processes category. Several studies propose an overall approach and/or outline a process for automotive software product line development. In some cases, specific methods and techniques are proposed, such that the respective studies are also assigned to other categories in Table 2.8.

Polzer et al. [92] present a framework for model-based automotive software product lines in which model transformations play an important role. The framework supports the creation of context-specific views, which provide a detailed description of the domain engineering and application engineering process tasks and artifacts. The approach also addresses the recovery and discovery of an evolved product line using a model-extraction process. Finally, authors define a product-derivation process for application engineering.

Hardung et al. [98] propose a framework to improve the reuse of automotive software. Their framework is based on the *Product Line Practice* (PLP; [15]) process model. Authors explain how to perform the modularization of the product line core assets, which are contained in a function repository. The process also defines how to develop products from that function repository using a standard software core. Finally, authors list tools to support the processes.

Tischer et al. [102] present experiences regarding the introduction od product lines at Bosch Gasoline Systems. Their study outlines the different relevant areas for SPL-based development, e.g., architecture development, product-line scoping and core asset development, market-oriented development, measurement of product line success, and product quality management. Furthermore, they discuss why the product line's deployment at Bosch has been delayed.

Lind and Heldal [83] and Eklund et al. [86] both propose an architecture-centered development approach and present detailed process descriptions. One focus of both studies is on reference architectures so that the studies are discussed in detail in category *reference architectures.*

In summary, the studies of this category describe an overall development approach for automotive software product line development, but at different levels of detail and with different focal points: the studies [83] and [86] are architecture-centered and contain more detailed process descriptions. In contrast, [102] remains fairly vague and does not contain a process description; instead, it provides an experience report. The other studies [92, 98] propose frameworks for software product line development: [92] focuses on model-based development using techniques like model transformation. The framework proposed in [98] allows for classifying software according to possible ways of reusing it, and it links the development process to the environment, i.e., repositories and tools. However, specific guidelines for developing automotive software product lines are not provided.

Compared to our classification schema (Section 1.4), neither approach [92, 98, 102, 83, 86] covers all activities. For instance, recovery & discovery is only supported by [92]. Yet, the framework in [92] does not address a product-specific development of implementation artifacts. Instead, implementation is only performed during domain engineering and serves as the basis for product derivation.

Reference Architectures

This section discusses the studies P43 [75], P90 [80], P129 [83], and P218 [86], which were assigned to the reference architecture category. Several studies propose reference architectures as a key element for the development of software product lines.

Martínez-Fernández et al. [75] present a survey on the benefits and drawbacks of using the software reference architecture AUTOSAR, an open industry standard for the automotive software architecture between suppliers and manufacturers. They conducted an online survey that addressed experienced AUTOSAR practitioners. They found standardization and reuse as most popular benefits, and complexity and initial investment as most remarked drawbacks and risks of using AUTOSAR.

Eklund and Bosch [80] proposed a reference architecture for embedded open software ecosystems consisting of 17 key decisions resulting in four architectural patterns. The architectural patterns are: *device abstraction*, *data and service provision*, *device and information composition*, and *safety-critical, certified and open application access.* These patterns have to be instantiated as a product line architecture for platform design. Furthermore, they define quality attributes for the reference architecture: *composability*, *deployability of new functions*, *stability over time*, *configurability*, *consistent user interface*, and *dependability.* Their approach is demonstrated with a prototypically implemented architecture that satisfies selected key decisions and quality attributes.

Lind and Heldal [83] study the research question *"How can a reference architecture for automotive systems be developed in a component-based setting, and how is it utilized in product* *development to increase reuse?"*. They focus on the definition of a development process with a reference architecture as a core artifact. The process is divided into E/E architecture development and E/E systems development. Their reference architecture is developed and maintained during E/E architecture development and serves as the basis for developing a product-specific architecture for E/E systems development. The proposed development process has been validated in several steps by projects at Saab Automobile AB. The result of the validation shows that the process works well at Saab.

Eklund et al. [86] propose an architecture-centered development process and describe how reference architectures are used to improve the development process. Authors describe the design, verification, dissemination, and maintenance of the reference architecture. Based on their experiences, they conclude that dissemination and maintenance require more resources than the development of the reference architecture. The proposed process results on the experiences with architecture-centered development at Volvo Cars.

Summarized, the studies [83] and [86] focus on the development process and how reference architectures are embedded into the development process. In contrast, the work of Eklund and Bosch [80] focus on the particular design of a reference architecture grounded in design decisions and patterns. The benefits of drawbacks of the AUTOSAR reference architecture are presented in [75]. As mentioned in [80], AUTOSAR assumes an integration-centric approach in which all integration work concerning software components is done by an OEM. Yet, authors of [80] argue that it is desirable to move away from integration-centric development of software towards a development in a open software ecosystem.

Software Product Line Merging

This section discusses the studies P94 [81], P173 [84], and P221 [88], which were assigned to the SPL merging category. If independently developed product lines, e.g., Diesel and Gasoline software systems, share a high degree of common functionalities, merging those products into one product line is reasonable.

The two studies by Yoshimura et al. [81, 84] primarily focus on effort estimation techniques for software product line merging (as already discussed in the category *cost/effort estimation*). Both studies also outline merging strategies. However, details about the actual process of performing the merge are missing in both studies.

Tischer et al. [88] present their experiences from merging two large-scale software product line development projects (a Diesel and Gasoline software system), which was motivated by the high synergy potential. The merge was performed at the different levels: *organization*, *software architecture*, *development environment*, and *processes and methods*. The study provides a comprehensive overview of the challenges and solutions chosen to merge these large-scale systems.

Testing/Verification

This section discusses the studies P310 [94] and P343 [96], which were assigned to the testing and verification category. Due to the high degree of variability that results in a potentially huge number of products, testing and verification of a software product line are challenging tasks.

Lochau et al. [94] present an approach of pairwise testing in the SPL context by providing a mapping between feature models and a reusable test model in the form of statecharts. Therefore, they investigate the relationship between feature-based coverage criteria and model-based coverage criteria. A further contribution is a reasoning about the applicability of this approach. The approach is validated by a case study from automotive software engineering.

Scheidemann [96] proposes an incremental process using approximation algorithms for minimizing the number of configurations needed to verify the completeness of the configuration space. For this purpose, the author introduces the concept of “locality sets” containing architectural elements that realize a functionality, which is concerned with a particular requirement. The proposed algorithms either choose the minimum set of configurations necessary to verify all requirements for all configurations, or to maximize the verification coverage of the software product line as a whole by choosing the “best” configurations. A further contribution is a technique for automatically determining commonalities in architecture and requirements.

Variability Management

This section discusses the studies P8 [73], P220 [87], P279 [90], P332 [95], P365 [97], P468 [100], P493 [101], P580 [103], P588 [104], and P589 [105], which were assigned to the (general) variability management category. Managing the variability within a software product line constitutes a challenging task and is addressed by nearly one third of the studies from the result set.

Jin Sun et al. [73] propose a component-based development process based on variability types. They identify several variability types that may occur in ECU-related development: variability of software components, variability of logic components, variability of sensor components, variability of actuator components, variability of setpoint generator components, variability of output device components, and variability of component interfaces. Furthermore, they propose a process for developing ECUs that focusses on managing the variability by using the introduced variability types.

Boutkova [87] present their experiences with variability management in the requirements specification process at Daimler passenger car development. Based on these experiences, Boutkova proposes a new feature-based variability management (FBVM) approach. This approach is extended by a decentralized variability modeling approach that supports variability modeling of individual components and systems as well as modeling a product as a whole.

Graf et al. [90] propose a graph-based approach for variant modeling and management. The approach focuses on managing, modeling, and combining the different kinds of knowledge in software product line development, i.e., combining local expert knowledge with domain knowledge provided by application groups, and the integration of overall management knowledge.

Brink et al. [95] propose an approach to link hardware and software variability models. Their approach distinguishes between software and hardware variants using separate variability models. They further distinguish two different kinds of dependencies between hardware- and software product lines.

Millo and Ramesh [97] propose an approach to link the design-level variability with the requirements-level variability. The requirements and designs are expressed as extended finite state machines, so called *Finite State Machines with Variability* (FSMv). The variability between designs and requirements is based on a conformance relation between design and requirements models of the software product line features. An algorithm is proposed for checking conformance between the models and is implemented on the verification tool SPIN.

Manz et at. [100] propose an approach for enabling traceability, integrated configuration of software variants, and links using integrated feature modeling. An integrated feature model considers the different development phases, abstraction layers, and development artifacts.

Kato and Yamaguchi [101] present an approach for variability management focused on pairwise feature interactions. They accumulate the occurrence of the feature interactions for all past products using a feature interaction matrix to visualize feature interactions.

Käßmeyer et al. [103] propose a process for a model-based change impact analysis. They use methods and techniques for safety analysis (see category *safety analysis* and configuration/variant management, and focus on the combination of both.

De Oliveira et al. [104] propose an approach to support safety analysis for software product lines. Their approach provides guidelines regarding the use of model-based development, safety analysis, and variability management tools. In [105], de Oliveira et al. also address safety engineering in software product lines. They propose a method to support automated construction of modular product line safety cases. The method uses outputs provided by model-based development, safety analysis, and variability management tools.

Summarized, variability management is addressed by the selected studies in different ways using different methods and techniques. Several studies deal with the modeling of variability (e.g., [73, 87, 90, 95, 97, 100, 101] and with the tracing of variability information across different levels of abstraction, like requirements and design [97], hardware and software [95], or safety analysis assets and design assets [103, 104, 105]. Three studies focus on the combination of variability management and safety analysis and are thus also classified in category *safety analysis* [103, 104, 105].

2.2.3 Summary and Discussion

We prepared the in-depth review by using a conceptual model for automotive specific software product line development [25] (see Figure 1.2) as well as the results of a scoping study [34] in the field of automotive software engineering. The in-depth review reveals that none of the studies represents a holistic approach for managed evolution of automotive software product lines according to the classification schema (Section 1.4). On average two activities of our classification schema are addressed by the selected primary studies (see Figure 2.3). In addition, according to the results of Section 2.2.2 most studies refer to activities related to product line development (domain engineering) including Recovery & Discovery (86%). The analysis of the methods and techniques in the previous section produces comparable results.

A further finding is the accumulation of studies addressing the first activities of the schema like the product line design. This indicates that the focus is on how to build up a product line, which is called *revolutionary architecture process* (RAP) in [85]. Axelsson [85] states that most literature mainly describes RAP, whereas *evolutionary architecture processes* (EAP) lack studies and evidence (see Section 2.2.2). Activities required for long term evolution (i.e., EAP according to [85]) like architecture conformance checking are only addressed by few studies.

Remarkably, there is a low number of contributions concerning the product specific development cycle, which may indicate a research gap or it may indicate, e.g., that only few specific methods and techniques are required for product development within a product line based development approach. However, further developments that are separate from the product line, have to be transferred back into product line development at a later stage. The corresponding activity `P to PL` of the conceptual model was solely subject of two studies [74, 81] according to the result of the categorization depicted in Figure 2.3. But the proposed methods and techniques of both studies [74, 81] have another focus (see Section 2.2.2) and do not support the actual objective of activity `P to PL`. Here, we hypothesis a further research gap which has to be addressed in future research towards a holistic approach for managed evolution of automotive software product line architectures.

3

Recovery and Discovery of Automotive Software Product Line Architectures

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 chapter, we investigate how approaches for architecture repair can be adapted to be applied to automotive software product line architectures.

At first, Section 3.1 summarizes a paper where we applied a methodical and structured approach of architecture restoration in the specific case of the brake servo unit (BSU). The paper sketches how we apply basic concepts of software product line extraction and software architecture re-engineering on automotive software product line development. As a prerequisite for repairing an eroded software, a validated set of product line requirements are needed. These product line requirements serve as the basis for a successful manageable and evolvable PLA, too. Thus, we propose activity diagrams in conjunction with an executable semantics for requirements validation in Section 3.2. Finally, in Section 3.3, we propose an approach for repairing an eroded software consisting of a set of product architectures by applying strategies for recovery and discovery of the product line architecture. Here, the focus is to provide a scalable approach that can be applied to a complete software status of an ECU whereas in Section 3.1 we only regard a single function.

3.1 Mastering Erosion of Software Architectures in Automotive Software Product Lines: A Case Study

This section summarizes:

Arthur Strasser, Benjamin Cool, Christoph Gernert, Christoph Knieke, Marco Körner, Dirk Niebuhr, Henrik Peters, Andreas Rausch, Oliver Brox, Stefanie Jauns-Seyfried, Hanno Jelden, Stefan Klie and Michael Krämer. 2014. Mastering Erosion of Software Architectures in Automotive Software Product Lines. In *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, ser. LNCS, vol. 8327. Springer, pp. 491–502.

In this paper we apply a methodical and structured approach of architecture restoration in the specific case of the BSU. Software product lines from existing BSU variants are extracted by explicit projection of the architecture variability and decomposition of the original architecture.

3.1.1 Background: Case Example BSU

Originally there was a general variant of the software for controlling the evacuation of the vacuum chamber of the BSU through the intake manifold. Later on a variant of BSU software was added, that featured an electric vacuum pump for the evacuation. The software variance was constituted by the presence or absence of a mechanical vacuum pump. When implementing the variability into software the developers chose the simplest and fastest way: Since the mechanical vacuum pump was installed only in diesel vehicles, the variance was realized by a query whether there is a gasoline or diesel engine. This query was already used in other vehicle functions.

This solution established itself over time, but was insufficient with the introduction of hybrid vehicles, as they may have both a gasoline engine and an electric vacuum pump. Therefore, the developers extended the initial "gasoline or diesel engine" query by another query, whether it is a hybrid vehicle. While this was purposeful to allow quick implementation of the BSU software for hybrid vehicles, it no longer corresponded to the original motivation whether a vacuum pump is available. Multiple implementations of such quick solutions in both actuator and sensor variance, led to a complex, hardly maintainable and extensible BSU software system. An extensive analysis and de novo establishment of a product line within the BSU software, an *architecture regeneration* was required.

3.1.2 Approach for Architecture Restoration

Our approach consists of two essential activities that are called architecture regeneration and long-term evolution (see Figure 3.1). The first activity aims to extract architectural significant concerns that led to design decisions in the eroded model artifacts. All extracted concerns are specified in the appropriate artifacts:

- *feature model*: Specification of corresponding vehicle variants. Associations to variation points within the use case model and template architecture document the affect of a configured variant in the model.
- *use case model*: Specification of functionality concerns as use cases of the application, which represent functional requirements.
- *template architecture*: Specification of the components as resulting elements of the appropriate design decisions. The template architecture provides a reusable application design for an appropriate configuration in the feature model.

Associations between the resulting artifacts are used to track applied design decisions in the template architecture. These tracks are considered for further development in a long term as illustrated in Figure 3.1.

In our case study, we specify two use case groups and two variation points that match the systems functionality. One use case group considers requirements that must be fulfilled by all ECU configurations (uses core BSU control application) and another group that must only be realized by some ECU configurations. The variation points that extend (modify) a certain functionality at a defined point of behavior are associated with two features of the feature model. The two feature-groups group other features that represent different variants of evacuation methods and sensors. The group is associated with a variation point within the use case "control pressure difference". Every grouped feature can be independently selected for a valid ECU configuration. Thereby we documented the extensibility requirements as variation points within the two use cases.

The extracted product line requirements include significant design constraints that must be considered by the architect. All components that are associated with the kernel use cases realize

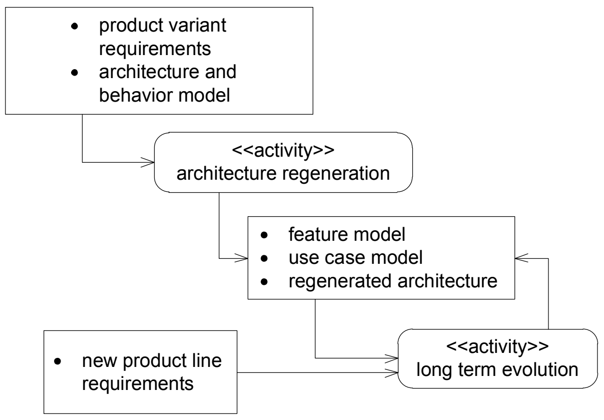

• product variant requirements
• architecture and behavior model
<<activity>> architecture regeneration
• feature model
• use case model
• regenerated architecture
• new product line requirements
<<activity>> long term evolution

Fig. 3.1. Required activities for mastering erosion in software product line

the functionality BSU pressure control. All components that are associated with extended use cases interact with the BSU pressure control through generic interfaces. Moreover a certain extended use case is associated with components that realize the appropriate sensor concern. These design decisions can be tracked and verified against further modifications to obtain a modular design. This enables the interconnect of all components according to feature selection. We call the final design “template architecture”.

A design decision that would cause additional dependencies between components associated to different variation points of the BSU, violates the modularity. Our approach establishes a relation between the variability models (functionality model) and the template architecture (architecture model) by applying the extraction phase. In the BSU real world example all use cases are one-to-one mapped to variation point features. The resulting feature model, use case model, and template architecture are depicted and explained in the paper.

3.2 Simulation and Validation of Executable Requirements Specification using Activity Diagrams

This section summarizes:

Christoph Knieke, Björn Schindler, Ursula Goltz and Andreas Rausch. 2012. Defining Domain Specific Operational Semantics for Activity Diagrams. In *IfI Technical Report Series*, IfI-12-04, Department of Informatics, TU Clausthal.

In this paper, we analyze common semantical constructs and possible options for the definition of operational semantics for activity diagrams. Activity diagrams are supported by a number of tools enabling not only the modeling but also the execution of activity models. Thereby, models can be verified and validated by simulation. Tools for such issues have to meet various requirements, depending on the domain of the system under development. For instance, the user of a simulator may validate the model in various ways. In information systems (e.g., big financial systems) the data processed by the system may be observed and various execution paths may be tested step by step. Therefore, the user has to be involved in system decisions. Consequently the system has to stop at control nodes (see Figure 3.2). At reactive systems (e.g., automotive ECU software) the behavior of external systems may be used for validation. The simulation has to realize, for instance, an exact concurrent execution. At the process of Figure 3.2 actions A2 and A4 are modeled to be executed concurrently. Hence, connected control nodes may have to be processed at once.

This example shows that tools need an adequate interpretation of the UML activity diagram semantics, which goes beyond the provided options of the UML superstructure [111]. Nevertheless, we observed that these semantics are based on the same semantical constructs. Depending on the domain, variants of these semantical constructs are used. To achieve an understanding of the semantics of activity diagrams independently from the domain, common semantical constructs as well as the variants have to be analyzed. This enables unification and tool support for defining domain specific operational semantics for UML 2 activity diagrams.

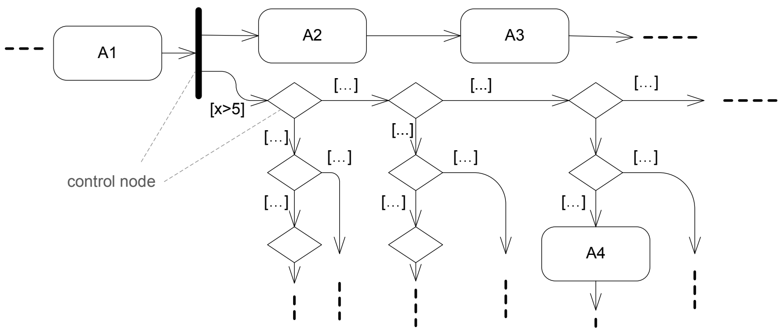


Fig. 3.2. Complex system decisions

We propose a foundation for a framework that enables composition of operational semantics for activity diagrams. The composition is based on fundamental semantic constructs with options enabling the definition of domain specific variants. By our approach, a clear and intuitive operational semantics for tool development can be composed, which conforms to the informal semantics description of the UML.

As an example, we introduce two tool developments based on particular operational semantics composed in our approach. In both cases, activity diagrams are considered for the specification of system requirements during the early development phase, which is used for validation. The semantics as well as the tools are well evaluated at research projects and used for realistic system developments. The first tool environment focuses on the modeling of information system behavior [112] whereas the second tool aims at the modeling of reactive system behavior [113].

3.3 From Product Architectures to a Managed Automotive Software Product Line Architecture

This section summarizes:

Benjamin Cool, Christoph Knieke, Andreas Rausch, Mirco Schindler, Arthur Strasser, Martin Vogel, Oliver Brox and Stefanie Jauns-Seyfried. 2016. 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. ACM, pp. 1350–1353.

In this paper 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. 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 an eroded PA 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* phase we recover the APLA and discover a TPLA candidate.

In the *Recovery & Discovery* phase we are using domain specific expertise and architecture related data from a repository to create the two PLAs. Figure 3.3 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

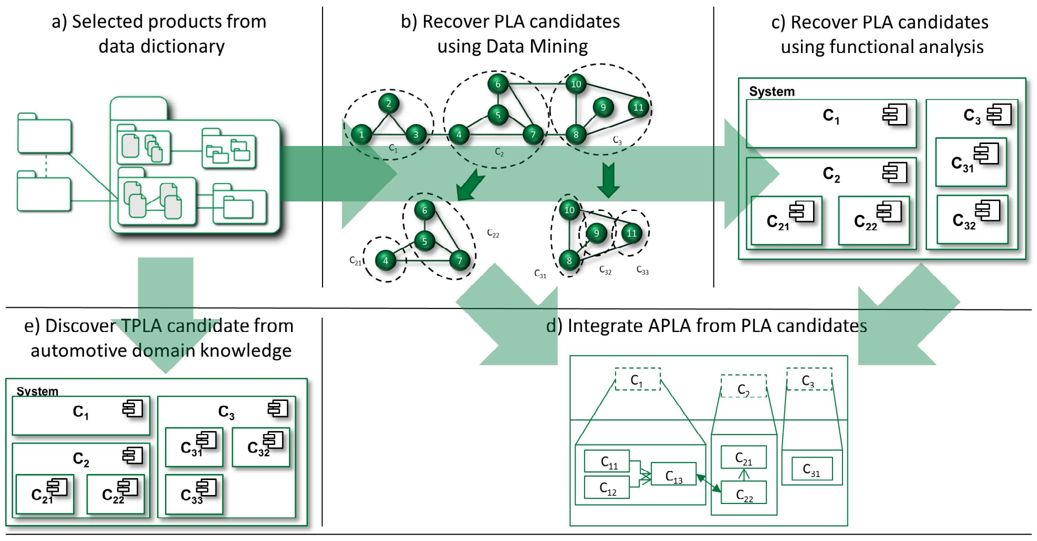

a) Selected products from data dictionary
b) Recover PLA candidates using Data Mining
c) Recover PLA candidates using functional analysis
d) Integrate APLA from PLA candidates
e) Discover TPLA candidate from automotive domain knowledge
System
C1
C2
C3
C21
C22
C31
C32
C33
C11
C12
C13

Fig. 3.3. Overview of phase Recovery & Discovery

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. For example, we only select the most recent created module and project versions.

b) **Recover APLA 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) [114, 115, 116] and is a generic approach to cluster software elements by their dependencies.
The SPAA approach is divided into three steps: 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 [115]. 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 [114] and [115].
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 [114] and [115]. The application of Spectral Clustering results in a cluster separation of the weighted graph, as presented in [115] 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 an APLA of the selected products including a hierarchical grouping of modules and indicating the inter group dependencies.

c) **Recover APLA candidates using functional analysis:** The aim of this step is to recover an APLA candidate using a technique considering the functionality aspects. In the ECU software development most of them are open/closed control loop related functions [117, 118]. At first we create a number of processing function related groups 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, which 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 APLA candidate, which consists of several hierarchical groups and group dependencies.

d) **Integrate APLA from APLA candidates:** The steps *b)* and *c)* produce APLA 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 Chapter 4). Therefore the integration of all available APLA candidates is necessary. We propose two essential steps for integration: At first groups are created representing the leafs of the APLA. Therefore the appropriate groups of the APLA 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 APLA candidates. The resulting groups are determined again by a comparison in an evaluation step. The second step is applied iteratively for each available APLA candidate level.

e) **Discover TPLA from automotive domain knowledge:** As an initial starting point for the following managing activities (see Chapter 4) a TPLA is needed. A TPLA contains at least the planned structure compared to the APLA. This knowledge has to be identified by 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 [119, 120]. The TPLA is created iteratively considering the knowledge of experts.

4

A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures

4.1 Overall Development Approach

This section summarizes:

Christoph Knieke, Marco Körner, Andreas Rausch, Mirco Schindler, Arthur Strasser and Martin Vogel. 2017. A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures. In *Special Track: Managed Adaptive Automotive Product Line Development (MAAPL), along with ADAPTIVE 2017*, IARIA XPS Press, pp. 43–52.

As discussed in [25] classical holistic approaches on software product line engineering have to be adapted to the special requirements of the automotive domain: In the automotive sector, it is not possible to carry out all further developments within the product line. Rather, there may be further developments that do not take place in the product line but at the level of the individual products. The reasons for this may be the high time and cost pressure, but also the fact that sophisticated further developments are initially to be tested within the scope of a prototype implementation. These further developments, which are separate from the product line, have to be transferred into product line development at a later stage. In addition, we often have to deal with an initially eroded software architecture which first has to be repaired as described in Chapter 3.

A challenge is to minimize architecture erosion in the long term: The product line architecture is designed initially and develops over time. In the further development, it must be ensured that the product architecture remains compliant with the product line architecture. Thus, in order to prevent architecture erosion in the future, architecture conformance checking is required for all further developments.

In the following we will explain the basic activities of our approach as already introduced in Section 1.4 and depicted in Figure 1.2 (see page 9). We have thematically grouped the activities:

1. Activity `Recovery & Discovery` aims at repairing an eroded architecture as described in the previous chapter.
2. Activities `PL-Requirements`, `P to PL`, `PL-Design` and `PL-Plan` are used for planning and evolving automotive software product line architectures. The focus is on the PLA design and planning of development iterations.
3. Activities `PL-Implement`, `P-Requirements`, `PL to P`, `P-Plan` and `P-Implement` address automotive product development and prototyping. These activities describe the implementation

aspects on the two levels – product line development and product development – including testing and prototyping.

4. Finally, activities `PL-Check`, `P-Design` and `P-Check` deal with architecture conformance checking. The aim is to limit product architecture erosion by techniques for monitoring architecture conformance.

We briefly introduce the activities in the following using these thematic groups (for group 1., see Chapter 3). Afterwards, Sections 4.2, 4.3 and 4.4 summarize several papers referring to the same thematic groups for the activities. Table 4.1 gives a brief overview on the objectives of each of the activities, including inputs and outputs.

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

2. Planning and Evolving Automotive Software Product Line Architectures

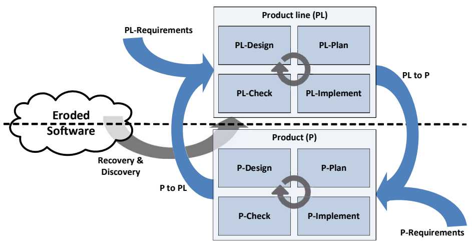
(`PL-Requirements`) Software system and software component requirements from requirements engineering serve as input to the management cycle of the PLA. Errors occurring during the phase of requirements elicitation and specification have turned out to be major reasons for the failure of IT projects [121]. In particular, errors occur in case the requirements are specified erroneous or the requirements have inconsistencies and incompleteness. Errors during the phase of requirements elicitation and specification can be avoided by choosing an appropriate specification language enabling the validation of the requirements. In [122], e.g., activity diagrams are considered for the validation of system requirements by directly executable models including an approach for symbolic execution and thus enabling validation of several products simultaneously.

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

(`PL-Design`) Next, we consider the design of the PLA. Generally, a software system architecture defines the basic organization of a system by structuring different architectural elements and relationships between them. The specification of "good" software system architecture is crucial for the success of the system to be developed. By our definition, a "good" architecture is a modular architecture, which is built according to the following: (a) design principles for high cohesion, (b) design principles for abstraction and information hiding, and (c) design principles for loose coupling. In [26], we propose methods and techniques for a good architecture design. Based on these methods and techniques a new PLA is defined (called PLA vision) taking the new requirements (`PL-Requirements`) and product related information (`P to PL`) into account. To assess the quality of the designed PLA, it is necessary to measure complexity and to describe the results numerically. In particular, we consider properties such as cohesion, coupling, reusability and variability in order to draw conclusions about the quality of the PLA.

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

Table 4.1. Explanation of the activities in Figure 1.2 (Section 1.4).



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

product developments, which must be taken into account when planning. Next, either affected projects and modules are determined or a pilot project is selected.

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

3. Automotive Product Development and Prototyping

(`PL-Implement`) The former planning activity has determined the schedule for PLA adaptations and product releases. Thus, on the implementation level, new versions of the software are planned, too. Vehicle functions are modeled using a set of modules, specifying the discrete and continuous behavior of the corresponding function. As required by ISO 26262 [123], each module needs to be tested separately. Established techniques for model-based testing necessitate a requirements specification, from which a test model can be derived. In practice, requirements are specified by natural language and on the level of whole vehicle functions instead of modules so that test models on module level can not be derived directly. Therefore, in [32], a systematic model-based, test-driven approach is proposed to design a specification on the level of modules, which is directly testable. The idea of test-driven development is to write a test case first for any new code that is written [124]. Then the implementation is improved to pass the test case. Based on the approach in [32] we use the tool Time Partition Testing (TPT) because it suits particularly well due to the ability to describe continuous behavior [125]. The modules may be developed in ASCET or MATLAB/Simulink.

(`P-Requirements`) Releasing a fully executable software status for a certain vehicle product requires a specification of vehicle related parameter settings. Furthermore, special requirements for a specific product may exist necessitating further development of certain implementation artifacts. Building an executable software status for a certain vehicle product is realized by the cycle at the bottom of Figure 1.2. In contrast, the product line cycle includes the development of sets of software artifacts of all planned projects.

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

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

(`P-Implement`) An iteration is completed when all planned elements of an iteration are implemented according to the test-driven approach of [32].

4. Architecture Conformance Checking

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

Each planned product refers to a set of implementation artifacts, called modules. These modules constitute the product architecture. The aim of `PL-Check` and `P-Check` is the minimization of product architecture erosion. In [64], a method is described to keep the erosion of the software to a minimum: Consistency constraints expressed by architectural aspects called architectural rules are specified as formulas on a common ontology, and models are mapped to instances of that ontology. Based on this approach we are extracting rules from a PLA to minimize the erosion of the product architecture. During the development of implementation artifacts the rules can be accessed via a query mechanism and be used to check the consistency of the product architecture. Those rules can, e.g., contain structural information about the software like allowed communications. In [64], the rules are expressed as logical formulas, which can be evaluated automatically to check the compliance with the PLA.

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

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

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

4.2 Planning and Evolving Automotive Software Product Line Architectures

We introduce an approach for automotive software systems evolution by concepts for planning and evolving product line architectures as already sketched in Section 4.1. First, we propose methods and concepts to create adequate architectures with the help of abstract principles, patterns, and describing techniques (see Section 4.2.1). Such techniques allow making complexity manageable. Next, we suggest techniques for the understanding of architecture and measuring of architecture quality (see Section 4.2.2). With the help of numerical results of these measurements, we can make a statement about complexity, as well as conclusions about a system. Furthermore, we describe how to plan development iterations and prototyping (see Section 4.2.3). Finally, we give an approach for extracting and managing architectural concepts (see Section 4.2.4).

The following three subsections 4.2.1, 4.2.2 and 4.2.3 summarize:

Axel Grewe, Christoph Knieke, Marco Körner, Andreas Rausch, Mirco Schindler, Arthur Strasser and Martin Vogel. 2017. 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, pp. 53–62.

4.2.1 Concepts for Designing Automotive Software Product Line Architectures

Design patterns, architectural patterns, and styles are an important and suitable means of specifying software architectures [55]. 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 constrains on how they can be combined [55].

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 [56].

In the same way, we illustrate some examples that we worked out in our automotive domain projects. Generally, 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 to point-to-point connections results 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 in our field studies. We present architectural concepts addressing this problem in particular.

The five presented architectural concepts in this paper 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 the representation technique supported by a uniform description language it was possible to reuse the concepts in other projects to increase the quality in an early phase of development.

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 studies 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.

Architectural concepts 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 [56]. 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 that are documented, evaluated, maybe extracted from existing products, but made explicit and integrated 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 [65]. Related to the fact that all elements of the software are subjects to the evolution process, architectural concepts can change or have 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.

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

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

In our approach, we assess the *modification effort*, *reuse rate* and *cohesion* of a PLA, since we can thus evaluate the properties discussed above. Furthermore, we give formulas and examples for the calculation of these metrics.

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. We exemplarily describe the procedure of applying change operations on a PLA.

4.2.3 Planning of Development Iterations and Prototyping

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

Planning consists of a sequence of iterations. Iterations are defined as a number of architecture elements that must be realized in a time period bounded by t_{start} and t_{end} with $t_{start}, t_{end} \in \mathbb{N}, t_{start} < t_{end}$. Within each time period the activities `Design`, `Plan`, `Implement` and `Check` are ordered. The iteration is completed when all modifications are realized by `Design`, `Implement`, and checked to be conform to architecture rules by `Check`. An example of a sequence of three iterations is shown in Figure 4.1: 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 4.1.

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 a modification realizing a new pattern in the appropriate PLA or PA. 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 project. This special case is dealt with in activity `PL to P` of our methodology.

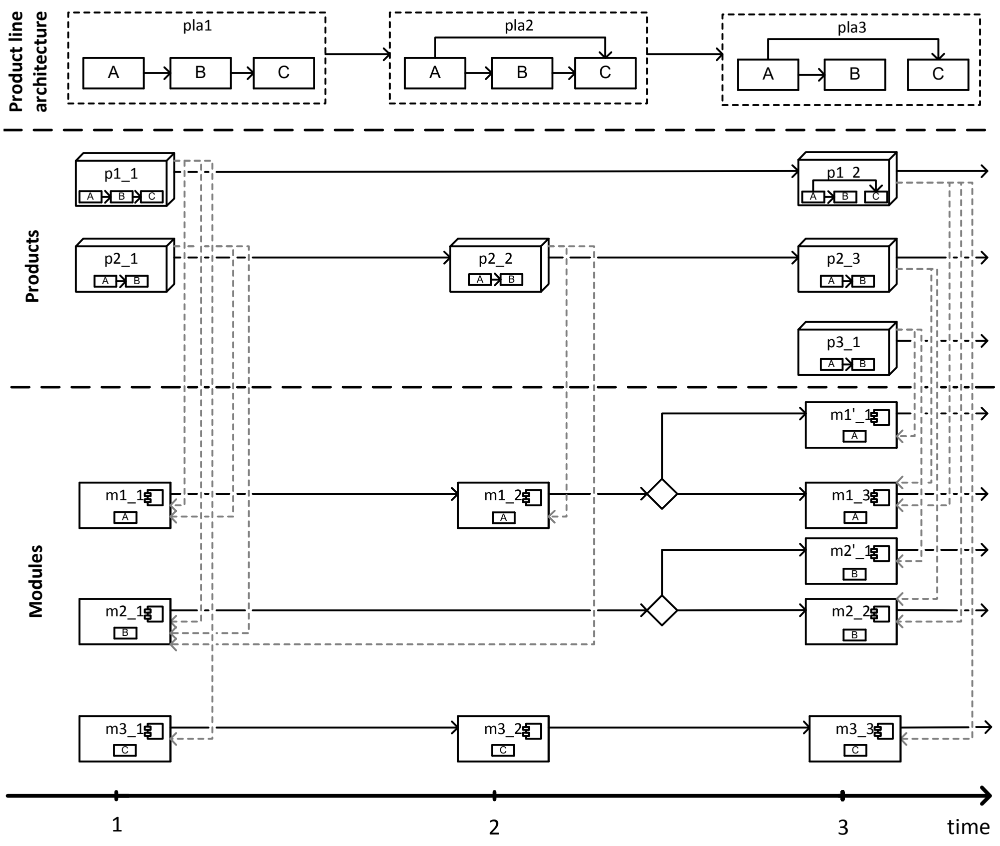


Fig. 4.1. Relation between PLA, products and modules

4.2.4 Knowledge-based Architecture Evolution and Maintenance

This section summarizes:

Axel Grewe, Christoph Knieke, Marco Körner, Andreas Rausch, Mirco Schindler, Arthur Strasser and Martin Vogel. 2017. Automotive Software Architecture Evolution: Extracting, Designing and Managing Architectural Concepts. In *International Journal On Advances in Intelligent Systems*, vol. 10, no. 3 & 4, pp. 203–222.

This paper is a substantial extension of the work in [30], summarized in the previous three subsections. We propose an approach to extract *architectural concepts* (cf. Section 4.2.1) for the design of automotive software product line architectures. Furthermore, we outline how this approach can be extended to a holistic solution for managing architectural concepts during the evolution of the system life-cycle.

Looking at different products and their architectures, the architectural concepts are very helpful to create a common product line architecture. For this reason, we focused on the research question how the developer's best practice can be identified and reflected to the architectural level. Figure 4.2 gives an overview on the approach to extract architectural concepts embedded into an evolutionary incremental development process.

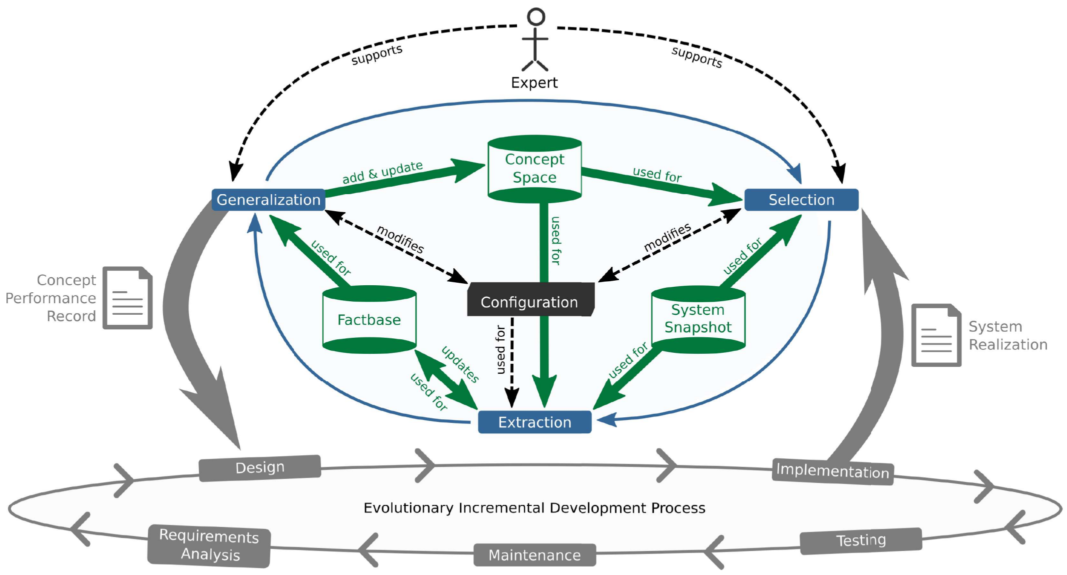


Fig. 4.2. Overview of the approach to extract architectural concepts embedded into an evolutionary incremental development process

To give a short introduction of the approach, we will give some general definitions of the terms used in Figure 4.2 in the following. 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 $e_i \in E$:

$$d_{p_j}(e_i) = \begin{cases} 1 & \text{, iff the Element } e_i \text{ fullfills the Property } p_j \\ 0 & \text{, otherwise} \end{cases} \tag{4.1}$$

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 on the realized systems. As shown in Figure 4.2 the extraction is based on 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 [65], [126] and [127].

Another data pool is the **Factbase $\mathfrak{F}$**, which represents the fulfillment of concepts for the concrete elements. It is divided into three parts, two data-structures organized in a table-structure listing facts referring to elements respectively to dependencies and one 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 4.2), 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, which are made in an activity.

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

Next, we summarize the selection, extraction and generalization activity: In the *selection activity* 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 *extraction activity* is fully automated and generates first the Factbase based for the selected elements by executing each detector for each element. After the Factbase has been enriched with new facts receptively potential new concepts, a validation of these facts is carried out in the *generalization activity* by an expert.

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 [128], Growing Neural Gas [129], and a Self-Organizing-Map (SOM) [130] orientated on the work of Matthias Reuter [131], [132]. These algorithms are used to find concepts on the system element level to detect special data-objects like TransferObjects [133], 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 [134],
2. Graph Clustering approaches like SPAA [114, 115, 116], and
3. t-SNE [135].

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.

As visualized in the bottom of Figure 4.2 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 realized concepts. This information can be combined with the results from the `PL-Check` and `P-Check` activity. 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 is 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 over time architectural concepts 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.

We demonstrate our approach on a real world example, the longitudinal dynamics torque coordination from automotive software engineering. In the case study we describe how we apply the approach to manage the complexity of the example system: First, we start with architecture recovery and extract essential architectural concepts. Then, within several iterations, we design the new PLA while continuously measuring architecture quality of each new design. We show that the application of the approach paves the way for long-term maintenance and extensible architectures.

4.3 Automotive Product Development and Prototyping

This section summarizes:

Henrik Peters, Christoph Knieke, Oliver Brox, Stefanie Jauns-Seyfried, Michael Krämer and Andreas Schulze. 2014. A Test-driven Approach for Model-based Development of Powertrain Functions. In *Agile Processes in Software Engineering and Extreme Programming. 15th International Conference on Agile Software Development, XP 2014*, Springer-Verlag, pp. 294–301.

Vehicle functions are modeled using a set of modules, specifying the discrete and continuous behavior of the corresponding function. As required by ISO 26262, each module needs to be tested separately. Established techniques for model-based testing necessitate a requirements specification, from which a test model can be derived. In practice, requirements are specified by natural language and on the level of whole vehicle functions instead of modules so that test models on module level can not be derived directly.

In this paper we propose a systematic model-based, test-driven approach to design a specification on the level of modules, which is directly testable (see Section 4.3.1). The idea of test-driven development is to write a test case first for any new code that is written [124]. Then the implementation is improved to pass the test case. In our approach we use the tool Time Partition Testing (TPT) because it suits particularly well due to the ability to describe continuous behavior [125]. The modules may be developed in ASCET or MATLAB/Simulink. Finally, we demonstrate our approach on a Selective Catalytic Reduction system, a real world example from automotive software engineering at Volkswagen (see Section 4.3.2).

4.3.1 Test-driven Approach

Continuous behavior and its testing have some specialties [136]. TPT is both a new testing methodology for testing continuous behavior of embedded systems in the automotive domain and a tool for supporting that methodology. TPT supports the activities modeling, execution, evaluation, and documentation of tests [125].

The platform-independent construction of test models is performed using a graphical, state-based notation. Entire sequences of test scenarios are decomposed to phases with states and transition conditions. To execute test cases automatically formal definitions are assigned to each element. All test cases of a scenario are derived from a single state machine using the classification tree method by combining the variation points.

The execution of test cases is platform-dependent by means of a test engine. To run the same test cases in a model-in-the-loop (MiL), a software-in-the-loop (SiL) or a hardware-in-the-loop (HiL) test, an abstract intermediate code is used. During execution all signals involved are recorded.

To evaluate the recorded signals properties have to be specified that must be met. The definition of a test oracle is difficult if the recorded signals are complex and often it is impossible to define the system's behavior based only on the outputs. Therefore, the recorded signals are abstracted in order to extract information and to allow general statements. For example, TPT enables the simple evaluation of threshold crossing or the simple comparison with reference signals. Any assessment can be valid globally or temporarily.

To start with our approach, some preparatory work is needed. The aim of the development is to implement a complete function of the engine control unit. However, currently our approach is limited on the development of a single module. Therefore, the function to be developed has to be decomposed into components, which are further divided into modules. For the decomposition of the function, an established procedure is used in the development process. During the first and the second decomposition, the respective interfaces have to be specified (between components and modules respectively). The result is a complete module architecture, in which each module can be designed.

We have adapted the test-driven development (TDD) cycle for model-based development, as well as enhanced it for model-based testing (MBT). Figure 4.3 shows the extended TDD cycle.

Each cycle starts with a requirement, which has to be implemented. Instead of writing a test directly, the test model is extended – based on criteria, which are described more detailed later. In this step, the necessary assesslets for specifying and checking the requirement and the necessary testlets to produce the desired input data for the system under test (SUT) are modeled. In the following the test cases are derived. TPT allows a manual selection of test cases and enables an automatic generation based on the classification tree method. The following steps are closely related to the original TDD approach, except that we refer to models instead of code.

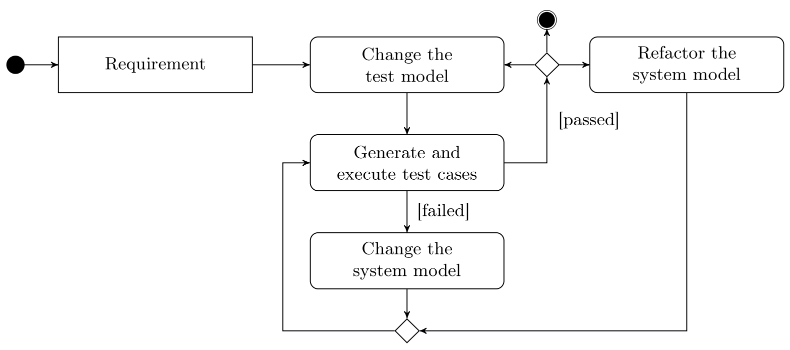


Fig. 4.3. Extended TDD cycle for model-based development

TDD focuses on unit tests to check the implementation. Since we use TDD to design a testable specification, we are focusing on acceptance tests. To ensure a high coverage of the generated code, we use coverage metrics like decision coverage for measuring the test quality.

Requirements are formulated with the aid of assesslets in TPT so that they can be viewed directly as a testable specification. Therefore, some properties have to be met. At module level the specification describes a required behavior. In order to observe behavior, signals are required, so that we define two rules:

1. We demand at least one requirement per signal.
2. We demand at least one signal per requirement.

It should be pointed out that the behavior of the respective signal may depend on other signals, characteristic values, curves, maps or system constants, which has to be considered.

In addition to these optional dependencies, a requirement has a unique name and via the documentation capabilities of TPT it is possible to add further descriptions (e.g., natural language comments, behavior or context diagrams, etc.). Furthermore, the conventional rules for requirements engineering apply (atomicity, consistency, etc.).

Each requirement must be covered by an acceptance test. According to the TDD cycle a new requirement may only be implemented if the corresponding acceptance test fails. This implies that defects that are not detected by a test have to be reproduced by a test before the adaptation of the system model is done [137]. Although acceptance tests are usually a kind of black-box testing, these rules also apply to local signals of the respective module. That means that the specification includes not only interface-related behavior. The test aspects of our approach can be classified as a modification of grey-box testing.

4.3.2 Case Study

We demonstrate our approach on a selective catalytic reduction (SCR) software system, a real world example from automotive software engineering at Volkswagen. The reduction of pollutant emissions is an important challenge in the automotive domain. An exhaust after-treatment system is the SCR. SCR is a means of reducing nitrogen oxides (NOx) contained in the exhaust emissions. The software of SCR is divided into different components, e.g., the heater, the pump, and the coordinator. The components itself are divided into modules. The pump component, for instance, consists among other things of a module for the pressure build-up after start, the controller, and the post-drive. In the paper we demonstrate the approach exemplarily on an excerpt of the module for the pressure build-up after start.

The interface of the module consists of an input, which provides the pressure in the reducing agent line. Outputs are the required mass flow and the required duty cycle needed for the pressure build-up. The starting point is an unspecified module except for the interface definition. Therefore, we begin to specify the behavior of the mass flow signal. The behavioral specification of the signal consists of four assesslets, where the first one specifies the valid range of the signal. The other three assesslets specify the behavior of the signal. During the specification of the output it turns out that the modules' behavior is dependent on an internal state of the module. Therefore, a new, local signal for the specification of this state is introduced. Further outputs are specified step-wise in the same manner.

To sum up, seven modules within the SCR functionality have been successfully developed. In addition, the test quality was evaluated by means of decision coverage. The domain models provide valuable information for the further integration of modules, e.g., for implementation in fixed-point arithmetic.

4.4 Architecture Conformance Checking

This section summarizes:

Christoph Knieke, Marco Körner, Andreas Rausch, Mirco Schindler, Arthur Strasser and Martin Vogel. 2017. Control Mechanisms for Managed Evolution of Automotive Software Product Line Architectures. In *International Journal On Advances in Software*, vol. 10, no. 3 & 4, pp. 191–210.

This paper is a substantial extension of the work in [25], summarized in Section 4.1 and presents an approach focusing on the minimization of architecture erosion in the automotive domain. We propose control mechanisms for a managed evolution of automotive software product line architectures. First, we introduce a description language and its meta model for the specification of the software product line architecture and the software architecture of the corresponding products (see Section 4.4.1). Based on the description language we propose an approach for architecture conformance checking to identify architecture violations as a means to prevent architecture erosion (see Section 4.4.2). We demonstrate our methodology on a real world case study, a brake servo unit (BSU) software system from automotive software engineering. To show the benefits of our approach, we define several metrics on architecture and software level and apply the metrics on the BSU example (see Section 4.4.3).

4.4.1 Architecture Description Language

The software architecture serves as input for the subsequent development steps, e.g., for implementation and test. In this architecture, the software building blocks of the cars embedded system are documented. Thereby, the implementation step follows the model-based development approach, where code is generated from architecture models using tools of the industrial partner. To model these architectures in the industrial projects we introduced the EMAB (Einheitliche Modulare Architektur Beschreibung) architecture description language. The EMAB is applied for the architecture extraction and the managed evolution approach presented in this paper. In addition, the EMAB includes all aspects to describe the static structure of the project partners electronic control unit system domains. EMAB is used to describe two layered architectures consisting of the logical architecture layer called DESIGN and the technical software architecture layer called IMPLEMENT. Both are defined by the syntax and semantics of the EMAB meta model elements. For each layer, the EMAB also defines the appropriate block diagram based views for architecture description. In our approach, the two layers DESIGN and IMPLEMENT refer to activities `PL-Design`/`P-Design` and `PL-Implement`/`P-Implement`, respectively.

In the paper, we illustrate the details of the EMAB syntax and appropriate semantics of the DESIGN layer and IMPLEMENT layer and their appropriate views. Furthermore, we show and explain the meta model in detail including the description language syntax and give an exemplary instance of the meta model.

From the technical point of view, the EMAB models are stored as XML files. During the export or import the file validity against the XML schema is checked. But not every model that is valid against the XML schema, is also a valid architecture description. Therefore, in the paper several rules are introduced for each layer in detail. These rules must be fulfilled to ensure the validity of the description of the two architecture layers.

4.4.2 Checking Views and Activities

In contrast to the validation checking of an EMAB model instance, where a syntax and general low-level semantic checking is performed, this subsection focuses on the individual product and the corresponding product line. The architectural concepts defined by the dedicated architecture and represented by its architectural rules are the input for this architecture conformance checking activity together with the implemented modules, respectively the realized parts of the systems. Output of a check is a set of violations, i.e., a list of pointers where the implementation does not fulfill the defined architecture.

As it is known from the field of architectural concepts and design patterns, these can be defined generally but it is not uncommon, that a software architecture contains more than one concept or pattern. Thereby the patterns can be adapted or modified to meet special requirements in a different level of specialization. Furthermore, the number of concepts and its variations are increasing steadily in practice. On the other hand, it might happen that realized concepts are dropped during the ongoing evolution steps. Hence, this activity has to be managed and supported by tools to support architects and developers, and for making concepts explicit.

We provide different views toward visualizing the architecture of a product and a product line. The EMAB meta model enables a mapping between elements of both views by the `maeMapping` association: Elements of the IMPLEMENT layer, the so called module architecture elements, are mapped to elements of the DESIGN layer, the so called logical architecture elements. The specified connections between the module architecture elements have to correspond to the connections between the mapped logical architecture elements. Otherwise, the checking activities will detect an architecture violation.

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

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

1. Creation of architecture rules,
2. selecting the implemented artifacts,
3. selecting the right rules,

4. performing the check and
5. visualizing the checking results to determine further actions.

For performing the checking activity we use an approach based on a logical fact base, as described in [64] and [127].

4.4.3 Evaluation and Discussion

To evaluate our methodology, we present the quantitative analysis for the BSU software development that is realized and maintained in cooperation with our project partner over a period of 5 years. In particular, we refer to the research objectives *2.a.* and *2.b.* as defined in Section 1.4:

1. Maintaining stability of the PLA and minimizing software product architecture erosion in real world automotive systems even if extensive further development of the system takes place.
2. High degree of reusability in real world automotive software development by achieving a high scalability, and a high degree of usage of the software components.

We could show that we have met these objectives. In the paper we first propose metrics for the evaluation criteria. Next, we apply our metrics to the BSU example. As a result, with regard to objective *2.a.*, we could limit architecture erosion to a minimum: Only one minor violation occurred in a period of five years. All further developments have followed the originally planned architectural principles and thus resulted in a high stability of the PLA. Moreover, with regard to objective *2.b.* we were surprised at the high number of usage of the modules: Most modules were used in all projects existing at that time. Only the scalability deteriorated in 2014 and 2015. But in 2016, the value has improved considerably again. Generally, a high degree of reuse could be observed: Each module was reused on average in 35 projects. Even the high number of potential variants could be managed with the approach.

5 Conclusions

In this thesis, I proposed an approach for managed evolution of automotive software product line architectures. Such a holistic approach is still missing as revealed by our systematic literature study. Moreover, the systematic literature study indicates a research gap in the field of the long term evolution of automotive software product line architectures.

As a first step of our methodology, we aim at repairing an eroded software architecture. Architecture repair involves the two approaches *recovery* and *discovery*. We show how these approaches can be adapted to recover the implemented automotive software product line architecture from the developed products and to discover the intended product line architecture.

Next, I proposed a holistic approach for a long-term manageable and plannable software product line architecture for automotive software systems. The approach aims to close the gap between product architectures and the product line architecture in the automotive domain. Thus, we use adapted concepts like architecture design principles, architecture compliance checking, and further development scheduling with specific adaptations to the automotive domain. The focus is on improving reusability and enabling scalability, to manage a huge number of variants in real world automotive systems.

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

In our activities for automotive product development and prototyping we address the implementation level, where amongst other things new versions of the software are planned. Vehicle functions are modeled using a set of modules, specifying the discrete and continuous behavior of the corresponding function. As required by ISO 26262, each module needs to be tested separately. Established techniques for model-based testing necessitate a requirements specification, from which a test model can be derived. In practice, requirements are specified by natural language and on the level of whole vehicle functions instead of modules so that test models on module level can not be derived directly. Therefore, we proposed a systematic model-based, test-driven approach to design a specification on the level of modules, which is directly testable.

Furthermore, we introduced an approach for architecture compliance checking of automotive software product line architectures. Modifications to a system that violate its architectural principles can degrade system performance and shorten its useful lifetime. As the potential frequency and scale of software adaptations increase to meet rapidly changing requirements and business conditions, controlling such architecture erosion becomes an important concern for software architects and developers. Therefore, we presented an approach focusing on the minimization of architecture erosion in the automotive domain. We introduced a description language for the specification of the logical product line architecture. Based on the description language we proposed an approach for architecture compliance checking to identify architecture violations as a means to prevent architecture erosion.

The presented methodology was derived and established in the context of several industrial automotive projects: These are a brake servo unit (BSU) software system, a selective catalytic reduction (SCR) software system and the longitudinal dynamics torque coordination. All of them are part of the engine control unit software at Volkswagen. Moreover, for the recovery and discovery phase we analyzed the complete software repository for the engine control unit software with 21,734 module versions. Some results of our work were even successfully used in series production at Volkswagen. As one result, we could limit architecture erosion to a minimum: Only one minor violation occurred in a period of five years. All other further developments have followed the originally planned architectural principles. Moreover, we were surprised at the high number of reuse of the modules: Each module was reused on average in 35 projects. Even the high number of potential variants could be managed with the approach. By the real world case studies, we could show that we have met all research objectives as defined in Section 1.4.

As a future work, we aim at realizing a tool-chain enabling the architecture description of the different architectures (product line architecture, product architecture, including versioning), the measure and evaluation of quality attributes, as well as the integration of the ArCh-Framework [64]. An associated challenge is the applicability of the tool-chain in a real world industrial project that targets the erosion management in the field of automotive product line architectures and product architectures used for automotive software sharing between the OEM and TIER1. Appropriate abstraction techniques are crucial to cope with the huge set of adjustable parameters within the ECU software and to manage variability. Thus, we are currently developing a concept including a prototypical tool environment that enables the description and visualization of variability.

References

1. J. Schäuffele and T. Zurawka, Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen, 4th ed. Vieweg+Teubner, 2010.
2. M. Broy, "Challenges in Automotive Software Engineering," in Proceedings of the 28th International Conference on Software Engineering, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 33–42.
3. J. Dokic, B. Müller, and G. Meyer, "European Roadmap Smart Systems for Automated Driving," European Technology Platform on Smart Systems Integration, 2015.
4. M. Broy, A. Pretschner, C. Salzmann, and T. Stauner, "Software-Intensive Systems in the Automotive Domain: Challenges for Research and Education ," SAE Technical Paper, Tech. Rep. 2006-01-1458, 2006.
5. 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.
6. V. Schulte-Coerne, A. Thums, and J. Quante, "Automotive Software: Characteristics and Reengineering Challenges," Softwaretechnik-Trends, vol. 29, no. 2, 2009.
7. A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in 2007 Future of Software Engineering, ser. FOSE '07. IEEE Computer Society, 2007, pp. 55–71.
8. McKinsey&Company, "Advanced Industry Report: Connected car, automotive value chain unbound," 2014. [Online]. Available: https://www.mckinsey.de/sites/mck_files/files/mck_connected_car_report.pdf
9. B. Boss, "Architectural Aspects of Software Sharing and Standardization: AUTOSAR for Automotive Domain," in Proceedings of the Second International Workshop on Software Engineering for Embedded Systems, ser. SEES '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 9–15.
10. E. Geisberger and M. Broy, Eds., agendaCPS - Integrierte Forschungsagenda Cyber-Physical Systems. Springer, 2012.
11. S. Thiel, M. A. Babar, G. Botterweck, and L. O'Brien, "Software Product Lines in Automotive Systems Engineering," SAE international journal of passenger cars-electronic and electrical systems, vol. 1, no. 2008-01-1449, 2008, pp. 531–543.
12. R. R. Macala, L. D. Stuckey, and D. C. Gross, "Managing Domain-Specific, Product-Line Development," IEEE Software, vol. 13, no. 3, May 1996, pp. 57–67.
13. M. H. Meyer and A. P. Lehnerd, The Power of Product Platforms. New York, NY: Free Press, 1997.
14. D. M. Weiss and C. T. R. Lai, Software Product-line Engineering: A Family-based Software Development Process. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
15. P. Clements and L. Northrop, Software Product Lines: Practices and Patterns. Addison Wesley, 2001.

16. K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005.
17. L. A. Belady and M. M. Lehman, "A Model of Large Program Development," IBM Systems journal, vol. 15, no. 3, 1976, pp. 225–252.
18. S. Cook, H. Ji, and R. Harrison, "Software Evolution and Software Evolvability," University of Reading, UK, 2000, pp. 1–12.
19. D. Rowe, J. Leaney, and D. Lowe, "Defining Systems Evolvability - A Taxonomy of Change," in International Conference and Workshop: Engineering of Computer-Based Systems (ECBS'98). IEEE Computer Society, April 1998, pp. 45–52.
20. H. P. Breivold, "Software Architecture Evolution and Software Evolvability," Ph.D. dissertation, Mälardalen University Sweden, School of Innovation, Design and Engineering, 2009.
21. L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 2nd ed. Addison-Wesley Professional, 2003.
22. 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.
23. F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," Computer, vol. 20, no. 4, Apr. 1987, pp. 10–19.
24. 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.
25. C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures," in Special Track: Managed Adaptive Automotive Product Line Development (MAAPL), along with ADAPTIVE 2017. IARIA XPS Press, 2017, pp. 43–52.
26. A. Rausch, O. Brox, A. Grewe, M. Ibe, S. Jauns-Seyfried, C. Knieke, M. Körner, S. Küpper, M. Mauritz, H. Peters, A. Strasser, M. Vogel, and N. Weiss, "Managed and Continuous Evolution of Dependable Automotive Software Systems," in Proceedings of the 10th Symposium on Automotive Powertrain Control Systems, 2014, pp. 15–51.
27. A. Strasser, B. Cool, C. Gernert, C. Knieke, M. Körner, D. Niebuhr, H. Peters, A. Rausch, O. Brox, S. Jauns-Seyfried, H. Jelden, S. Klie, and M. Krämer, "Mastering Erosion of Software Architecture in Automotive Software Product Lines," in SOFSEM 2014: Theory and Practice of Computer Science, ser. LNCS, V. Geffert, B. Preneel, B. Rovan, J. Stuller, and A. M. Tjoa, Eds., vol. 8327. Springer, 2014, pp. 491–502.
28. C. Knieke, B. Schindler, U. Goltz, and A. Rausch, "Defining Domain Specific Operational Semantics for Activity Diagrams," Department of Informatics, Technische Universität Clausthal, Tech. Rep. IfI-12-04, 2012.
29. C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "Control Mechanisms for Managed Evolution of Automotive Software Product Line Architectures," International Journal On Advances in Software, vol. 10, no. 3 & 4, 2017, pp. 191–210.
30. 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.
31. A. Grewe, C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "Automotive Software Architecture Evolution: Extracting, Designing and Managing Architectural Concepts," International Journal On Advances in Intelligent Systems, vol. 10, no. 3 & 4, 2017, pp. 203–222.
32. H. Peters, C. Knieke, O. Brox, S. Jauns-Seyfried, M. Krämer, and A. Schulze, "A Test-driven Approach for Model-based Development of Powertrain Functions," in Agile Processes in Software Engineering and Extreme Programming. 15th International Conference on Agile Software Development, XP 2014, G. Cantone and M. Marchesi, Eds. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 294–301.
33. C. Knieke and A. Rausch, "MAAPL: Managed Adaptive Automotive Product Line Development," 2017, in Editorial of Special Track MAAPL, along with the Ninth

International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2017), February 19 - 23, 2017 - Athens, Greece. [Online]. Available: https://www.iaria.org/conferences2017/filesADAPTIVE17/MAAPL_ADAPTIVE17.pdf

34. A. Haghighatkhah, A. Banijamali, O.-P. Pakanen, M. Oivo, and P. Kuvaja, "Automotive software engineering: A systematic mapping study," Journal of Systems and Software, 2017.
35. S. Clarke, B. Fitzgerald, P. Nixon, K. Pohl, K. Ryan, D. Sinclair, and S. Thiel, "The Role of Software Engineering in Future Automotive Systems Development," SAE International Journal of Passenger Cars-Electronic and Electrical Systems, vol. 1, no. 2008-01-1450, 2008, pp. 544–552.
36. K. Grimm, "Software technology in an automotive company: major challenges," in Proceedings of the 25th international conference on Software Engineering. IEEE Computer Society, 2003, pp. 498–503.
37. B. Gruszczynski, "An overview of the current state of software engineering in embedded automotive electronics," in Electro/information Technology, 2006 IEEE International Conference on. IEEE, 2006, pp. 377–381.
38. F. Fabbrini, M. Fusani, G. Lami, and E. Sivera, "Software engineering in the european automotive industry: Achievements and challenges," in Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International. IEEE, 2008, pp. 1039–1044.
39. K. Schmid, R. Rabiser, and P. Grünbacher, "A Comparison of Decision Modeling Approaches in Product Lines," in Proceedings of the 5th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'11), P. Heymans, K. Czarnecki, and U. W. Eisenecker, Eds. ACM, 2011, pp. 119–126.
40. K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches," in Proceedings of the 6th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'12). New York, NY, USA: ACM, 2012, pp. 173–182.
41. M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search Based Software Engineering for Software Product Line Engineering: A Survey and Directions for Future Work," in Proceedings of the 18th International Software Product Line Conference - Volume 1, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 5–18. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648513
42. E. Engström and P. Runeson, "Software product line testing–A systematic mapping study," Information and Software Technology, vol. 53, no. 1, 2011, pp. 2–13.
43. J. Lee, S. Kang, and D. Lee, "A Survey on Software Product Line Testing," in Proceedings of the 16th International Software Product Line Conference-Volume 1. ACM, 2012, pp. 31–40.
44. S. Oster, A. Wübbeke, G. Engels, and A. Schürr, "A Survey of Model-Based Software Product Lines Testing," Model-Based Testing for Embedded Systems, 2011, pp. 338–381.
45. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," ACM Computing Surveys (CSUR), vol. 47, no. 1, 2014, p. 6.
46. L. Chen, M. Ali Babar, and N. Ali, "Variability Management in Software Product Lines: A Systematic Review," in Proceedings of the 13th International Software Product Line Conference, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 81–90. [Online]. Available: http://dl.acm.org/citation.cfm?id=1753235.1753247
47. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature Diagrams: A Survey and a Formal Semantics," in Requirements Engineering, 14th IEEE international conference. IEEE, 2006, pp. 139–148.
48. 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.
49. 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.

50. 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.
51. G. Aldekoa, S. Trujillo, G. Sagardui, and O. Diaz, "Quantifying Maintainability in Feature Oriented Product Lines," in Proceedings of the 12th European Conference on Software Maintenance and Reengineering. IEEE, 2008, pp. 243–247.
52. T. Zhang, L. Deng, J. Wu, Q. Zhou, and C. Ma, "Some Metrics for Accessing Quality of Product Line Architecture," in 2008 International Conference on Computer Science and Software Engineering, vol. 2. IEEE, 2008, pp. 500–503.
53. 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.
54. H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, 2004.
55. M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
56. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing, 1996.
57. J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Pearson Education, 2000.
58. 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.
59. 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.
60. 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.
61. 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.
62. 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.
63. 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.
64. 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.
65. S. Herold, "Architectural Compliance in Component-Based Systems. Foundations, Specification, and Checking of Architectural Rules." Ph.D. dissertation, Technische Universität Clausthal, 2011.
66. J. van Gurp and J. Bosch, "Design Erosion: Problems & Causes," Journal of Systems and Software, vol. Volume 61, 2002, pp. 105–119.
67. B. A. Kitchenham, D. Budgen, and P. Brereton, Evidence-Based Software Engineering and Systematic Reviews. CRC Press, 2015.
68. K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," Inf. Softw. Technol., vol. 64, August 2015, pp. 1–18.
69. K. Petersen, R. Feldt, S. Mujtaba, and M. Mattson, "Systematic mapping studies in software engineering," in International Conference on Evaluation and Assessment in Software Engineering. ACM, 2008, pp. 68–77.
70. M. Kuhrmann, D. M. Fernández, and M. Daneva, "On the pragmatic design of literature studies in software engineering: an experience-based guideline," Empirical Software Engineering, Jan 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9492-y

71. M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," Empirical Software Engineering, vol. 16, no. 3, 2011, pp. 365–395.
72. B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University, Tech. Rep. EBSE-2007-01, 2007.
73. J. S. Her, S. W. Choi, D. W. Cheun, J. S. Bae, and S. D. Kim, "A Component-Based Process for Developing Automotive ECU Software," Lecture Notes in Computer Science, vol. 4589, 2007, p. 358.
74. M. Gleirscher, A. Vogelsang, and S. Fuhrmann, "A Model-Based Approach to Innovation Management of Automotive Control Systems," in Software Product Management (IWSPM), 2014 IEEE IWSPM 8th International Workshop on. IEEE, 2014, pp. 1–10.
75. S. Martínez-Fernández, C. P. Ayala, X. Franch, and E. Y. Nakagawa, "A Survey on the Benefits and Drawbacks of AUTOSAR," in Proceedings of the First International Workshop on Automotive Software Architecture. ACM, 2015, pp. 19–26.
76. S. Kiebusch, B. Franczyk, and A. Speck, "An Unadjusted Size Measurement of Embedded Software System Families and its Validation," Software Process: Improvement and Practice, vol. 11, no. 4, 2006, pp. 435–446.
77. R. Rana, M. Staron, C. Berger, J. Hansson, and M. Nilsson, "Analysing defect inflow distribution of automotive software projects," in Proceedings of the 10th International Conference on Predictive Models in Software Engineering. ACM, 2014, pp. 22–31.
78. M. Aoyama and A. Yoshino, "AORE (Aspect-Oriented Requirements Engineering) Methodology for Automotive Software Product Lines," in Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific. IEEE, 2008, pp. 203–210.
79. H. Gustavsson and U. Eklund, "Architecting Automotive Product Lines: Industrial Practice," Software Product Lines: Going Beyond, 2010, pp. 92–105.
80. U. Eklund and J. Bosch, "Architecture for Embedded Open Software Ecosystems," Journal of Systems and Software, vol. 92, 2014, pp. 128–142.
81. K. Yoshimura, D. Ganesan, and D. Muthig, "Assessing Merge Potential of Existing Engine Control Systems into a Product Line," in Proceedings of the 2006 international workshop on Software engineering for automotive systems. ACM, 2006, pp. 61–67.
82. J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortés, "Automated Diagnosis of Feature Model Configurations," Journal of Systems and Software, vol. 83, 07 2010, pp. 1094–1107.
83. K. Lind and R. Heldal, "Automotive System Development using Reference Architectures," in Software Engineering Workshop (SEW), 2012 35th Annual IEEE. IEEE, 2012, pp. 42–51.
84. K. Yoshimura, D. Ganesan, and D. Muthig, "Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems," in Proceedings of the 6th ACM & IEEE International conference on Embedded software. ACM, 2006, pp. 63–72.
85. J. Axelsson, "Evolutionary Architecting of Embedded Automotive Product Lines: An Industrial Case Study," in Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on. IEEE, 2009, pp. 101–110.
86. U. Eklund, O. Askerdal, J. Granholm, A. Alminger, and J. Axelsson, "Experience of Introducing Reference Architectures in the Development of Automotive Electronic Systems," in Proceedings of the Second International Workshop on Software Engineering for Automotive Systems, ser. SEAS '05. New York, NY, USA: ACM, 2005, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083195
87. E. Boutkova, "Experience with Variability Management in Requirement Specifications," in Software Product Line Conference (SPLC), 2011 15th International. IEEE, 2011, pp. 303–312.
88. C. Tischer, A. Muller, T. Mandl, and R. Krause, "Experiences from a Large Scale Software Product Line Merger in the Automotive Domain," in Software Product Line Conference (SPLC), 2011 15th International. IEEE, 2011, pp. 267–276.
89. D. Merschen, A. Polzer, G. Botterweck, and S. Kowalewski, "Experiences of Applying Model-based Analysis to Support the Development of Automotive Software Product Lines," in Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. ACM, 2011, pp. 141–150.

90. S. Graf, M. Glaß, D. Wintermann, J. Teich, and C. Lauer, “IVaM: Implicit Variant Modeling and Management for Automotive Embedded Systems,” in Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on. IEEE, 2013, pp. 1–10.
91. A. Leitner, N. Kajtazovic, R. Mader, C. Kreiner, C. Steger, and R. Weiß, “Lightweight introduction of EAST-ADL2 in an automotive software product line,” in System Science (HICSS), 2012 45th Hawaii International Conference on. IEEE, 2012, pp. 5526–5535.
92. A. Polzer, D. Merschen, G. Botterweck, A. Pleuss, J. Thomas, B. Hedenetz, and S. Kowalewski, “Managing complexity and variability of a model-based embedded software product line,” Innovations in Systems and Software Engineering, vol. 8, no. 1, 2012, pp. 35–49.
93. S. Wang, “Model Transformation for High-Integrity Software Development in Derivative Vehicle Control System Design,” in High Assurance Systems Engineering Symposium, 2007. HASE’07. 10th IEEE. IEEE, 2007, pp. 227–234.
94. M. Lochau, S. Oster, U. Goltz, and A. Schürr, “Model-based pairwise testing for feature interaction coverage in software product line engineering,” Software Quality Journal, vol. 20, no. 3, Sep 2012, pp. 567–604.
95. C. Brink, E. Kamsties, M. Peters, and S. Sachweh, “On Hardware Variability and the Relation to Software Variability,” in Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on. IEEE, 2014, pp. 352–355.
96. K. D. Scheidemann, “Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems,” in Software Product Line Conference, 2006 10th International. IEEE, 2006, pp. 75–84.
97. J.-V. Millo and S. Ramesh, “Relating Requirement and Design Variabilities,” in Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, vol. 2. IEEE, 2012, pp. 35–42.
98. 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.
99. J. E. Kim, R. Kapoor, M. Herrmann, J. Haerdtlein, F. Grzeschniok, and P. Lutz, “Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development,” in Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. IEEE, 2008, pp. 307–311.
100. C. Manz, M. Stupperich, and M. Reichert, “Towards Integrated Variant Management in Global Software Engineering: An Experience Report,” in Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on. IEEE, 2013, pp. 168–172.
101. S. Kato and N. Yamaguchi, “Variation Management for Software Product Lines with Cumulative Coverage of Feature Interactions,” in Software Product Line Conference (SPLC), 2011 15th International. IEEE, 2011, pp. 140–149.
102. C. Tischer, A. Muller, M. Ketterer, and L. Geyer, “Why does it take that long? Establishing Product Lines in the Automotive Domain,” in Software Product Line Conference, 2007. SPLC 2007. 11th International. IEEE, 2007, pp. 269–274.
103. M. Käßmeyer, M. Schulze, and M. Schurius, “A process to support a systematic change impact analysis of variability and safety in automotive functions,” in Proceedings of the 19th International Conference on Software Product Line. ACM, 2015, pp. 235–244.
104. A. L. de Oliveira, R. T. Braga, P. C. Masiero, Y. Papadopoulos, I. Habli, and T. Kelly, “A Model-Based Approach to Support the Automatic Safety Analysis of Multiple Product Line Products,” in Computing Systems Engineering (SBESC), 2014 Brazilian Symposium on. IEEE, 2014, pp. 7–12.
105. ——, “Supporting the Automated Generation of Modular Product Line Safety Cases,” in Theory and Engineering of Complex Systems and Dependability. Springer, 2015, pp. 319–330.
106. H. Gustavsson and J. Axelsson, “Evaluating Flexibility in Embedded Automotive Product Lines Using Real Options,” in Software Product Line Conference, 2008. SPLC’08. 12th International. IEEE, 2008, pp. 235–242.
107. R. Wieringa, N. Maiden, N. Mead, and C. Rolland, “Requirements engineering paper classification and evaluation criteria: A proposal and a discussion,” Requirements Engineering, vol. 11, no. 1, Dec. 2005, pp. 102–107.

108. J. L. Fleiss, "Measuring nominal scale agreement among many raters," Psychological Bulletin, vol. 76, no. 5, 1971, pp. 378–382.
109. M. Amram and N. Kulatilaka, Real options. Harvard Business School Press Boston, Massachusetts, 1999.
110. T. Copeland and V. Antikarov, Real Options: A Practitioner's Guide. TEXERE New York, 2001.
111. OMG, "UML, Version 2.5, OMG Specification Superstructure and Infrastructure," 2015. [Online]. Available: http://www.omg.org
112. M. Deynet, S. Niebuhr, A. Rausch, and B. Schindler, "Enhancing Validation with Prototypes out of Requirements Models," in ASWEC '10: 21st Australian Software Engineering Conference, Industry Track. IEEE Computer Society, 2010.
113. C. Knieke and U. Goltz, "An Executable Semantics for UML 2 Activity Diagrams," in ECOOP 2010 Workshop Proc. of the Intern. Workshop on Formalization of Modeling Languages (FML 2010). ACM Press, 2010, pp. 11–15.
114. M. Schindler, "Automatische Identifikation und Optimierung von Komponentenstrukturen in Softwaresystemen," Master's thesis, Technische Universität Clausthal, 2010.
115. 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.
116. 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.
117. 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.
118. 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.
119. K. Reif, Automobilelektronik - Eine Einführung für Ingenieure, 4th ed. Vieweg + Teubner, 2012.
120. R. Isermann, Ed., Elektronisches Management motorischer Fahrzeugantriebe, 4th ed. Vieweg + Teubner, 2010.
121. The Standish Group International, Inc., "CHAOS Chronicles 2003 report," West Yarmouth, MA, 2003.
122. C. Knieke and M. Huhn, "Semantic Foundation and Validation of Live Activity Diagrams," Nordic Journal of Computing, vol. 15, no. 2, 2015, pp. 112–140.
123. International Organization for Standardization, "ISO/DIS 26262: Road vehicles – functional safety," 2009.
124. K. Beck, Test Driven Development. By Example. Addison-Wesley Longman, 2002.
125. E. Lehmann, "Time Partition Testing – Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen," Ph.D. dissertation, Fakultät IV – Elektrotechnik und Informatik, TU Berlin, 2004.
126. C. Deiters, Beschreibung und konsistente Komposition von Bausteinen für den Architekturentwurf von Softwaresystemen, 1st ed., ser. SSE-Dissertation. München: Dr. Hut, 2015, vol. 11.
127. M. Mues, "Taint Analysis - Language Independent Security Analysis for Injection Attacks," Master's Thesis, Technische Universität Clausthal, Institute for Applied Software Systems Engineering, 2016.
128. M. Cottrell, B. Hammer, A. Hasenfuß, and T. Villmann, "Batch and median neural gas," Neural Networks, vol. 19, no. 6, 2006, pp. 762–771.
129. 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.
130. T. Kohonen, "The self-organizing map," Neurocomputing, vol. 21, no. 1, 1998, pp. 1–6.
131. 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.

132. 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.
133. A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, Eds., The Common Component Modeling Example: Comparing Software Component Models. Springer, 2008, vol. 5153.
134. 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.
135. 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.
136. E. Bringmann, “Testing the Continuous Behavior of Embedded Systems,” in Proceedings of the 4th Workshop on System Testing and Validation, 2007.
137. T. Dohmke, “Test-Driven Development of Embedded Control Systems: Application in an Automotive Collision Prevention System,” Ph.D. dissertation, Department of Mechanical Engineering, Faculty of Engineering, University of Glasgow, 2008.