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

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

Abstract—**The automotive industry aspires a high degree of reuse in software development in order to reduce the development costs. The reuse is achieved by a product-wide development for different vehicle variants, as well as by reuse in subsequent products. However, the increasing complexity and degree of variability of automotive software systems hinders the capabilities for reusability and extensibility of these systems to an increasing degree. After several product generations, software erosion is growing steadily, resulting in an increasing effort of reusing software components, and planning of further development. Here, we give 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. We demonstrate our methodology on a real world case study, a brake servo unit (BSU) software system from automotive software engineering.**

Keywords*–*Architecture Evolution; Software Product Lines; Software Erosion; Architecture Compliance Checking; Automotive.

I. INTRODUCTION

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

However, the possibilities for reuse and extensibility of existing functions can not be fully exploited in many cases. Rather, it can be observed that due to the increase in so-called "accidental" complexity [2] (see Section V-B), the reusability and further developability reaches its limits. One reason for this is the lack of a product-line-oriented overall planning, based on the concepts of software product line engineering already established in other domains. A central factor here is the planning based on a product line architecture (PLA), on the specification of which the individual products are derived. The PLA describes the structure of all realizable products. Each product that is developed has an individual product architecture (PA) whose structure should be mapped onto the PLA.

However, an overall specification of a PLA is often missing in the automotive domain [3]: The knowledge of the overall, product independent structure is not explicitly documented, and therefore exists only implicitly in the minds of the participants. Here, we refer to the results shown in a preceding paper [3] to create a PLA as a prerequisite for our approach by applying strategies for architecture recovery and discovery.

However, the application of the software product line development must take into account the special properties, boundary conditions and requirements that exist in the automotive environment [4]. Therefore, a method adapted to the automotive environment is required and is presented in this paper.

An important aspect is the design and planning of further developments of the product line architecture. When designing the product line architecture, the architecture must be based on architecture principles appropriate for the automotive domain, aiming at reusability and further development [2]. Since a wide range of products can be affected by the further development of the product line architecture, changes must be carefully planned: High demands are placed on the reliability of the systems, but the reliability is endangered by extensive adaptations.

In the further development, it must be ensured that the product architecture remains compliant with the product line architecture. However, due to the high time and cost pressure in the automotive sector, it is not possible, for every further development to be controlled via the product line. Rather, some product-specific adjustments have to be made. This can lead (intentionally or unintentionally) to a product architecture that differs in comparison to the product line architecture: the architecture erodes. In the long term, this leads to reduced reusability and extensibility of the software artifacts. Due to the size of the product line architecture, an automated consistency check is necessary, which is an essential part of our approach to counteract architectural erosion.

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

- Long-term minimization of architecture erosion.
- High degree of reusability.
- Scalability to manage a huge number of variants in real world automotive systems.

The paper is structured as follows: Section II gives an overview on the related work. In Section III, we propose a methodology for managed evolution of automotive software product line architectures. Section IV introduces parts of the architecture description language, which we will refer to in the

following sections. In Section V, we apply our approach on a real world example, a brake servo unit, from automotive software engineering. The results of a corresponding field study are evaluated and discussed in Section VI. Section VII concludes.

II. Related Work

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

A. Reference Architectures

The purpose of the reference architecture is to provide guidance for future developments. In addition, the reference architecture incorporates the vision and strategy for the future. The work in [5] examines current reference architectures and the driving forces behind development of them to come to a collective conclusion on what a reference architecture should truly be. Furthermore, in [5], reference architectures are assumed to be the basis for the instantiation of product line architectures (so-called family architectures, see [5]).

Nakagawa et. al. discuss the differences between reference architectures and product line architectures by highlighting basic questions like definitions, benefits, and motivation for using each one, when and how they should be used, built, and evolved, as well as stakeholders involved and benefited by each one [6]. Furthermore, they define a reference model of reference architectures [7], and propose a methodology to design product line architectures based on reference architectures [8][9].

B. Software Erosion

In [10], de Silva and Balasubramaniam provide a survey of technologies and techniques either to prevent architecture erosion or to detect and restore architectures that have been eroded. However, each approach discussed in [10] refers to architecture erosion for a single PA, whereas architecture erosion in software product lines are out of the scope of the paper. Furthermore, as discussed in [10], none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

Van Gurp and Bosch [11] illustrate how design erosion works by presenting the evolution of the design of a small software system. The paper concludes that even an optimal design strategy for the design phase does not lead to an optimal design. The reason for this are unforeseen requirement changes in later evolution cycles. These changes may cause design decisions taken earlier to be less optimal.

The work in [12] describes an approach to flexible architecture erosion detection for model-driven development approaches. Consistency constraints expressed by architectural aspects called architectural rules are specified as formulas on a common ontology, and models are mapped to instances of that ontology. A knowledge representation and reasoning system is then utilized to check whether these architectural rules are satisfied for a given set of models. Three case studies are presented demonstrating that architecture erosion can be minimized effectively by the approach.

C. Software Product Line Architectures

As discussed in [3] an overall automotive product line architecture is often missing due to software sharing. Thus, architecture recovery and discovery has to be applied by concepts of software product line extraction [3]. 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 [13] 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 [14]. 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.

In numerous publications, Bosch et. al. address the field of product line architecture, software architecture erosion, and reuse of software artifacts: The work in [16] proposes a method that brings together two aspects of software architecture: the design of software architecture and software product lines. Deelstra et al. [17] provide a framework of terminology and concepts regarding product derivation. They have identified that companies employ widely different approaches for software product line based development and that these approaches evolve over time. The work in [18] discusses six maturity levels that they have identified for software product line approaches. In [19], a methodical and structured approach of architecture restoration in the specific case of the brake servo unit (BSU) is applied. Software product lines from existing BSU variants are extracted by explicit projection of the architecture variability and decomposition of the original architecture.

The work in [20] gives a systematic survey and analysis of existing approaches supporting multi product lines and a general discussion of capabilities supporting multi product lines in various domains and organizations. They define a multi product line (MPL) as a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system. The different product lines in an MPL can exist independently but typically use shared resources to meet the overall system requirements. According to this definition, a vehicle system is also an MPL assuming that each product line is responsible for a particular subsystem. However, in the following, we only regard classic product lines, since the dependencies between the individual product lines in vehicle systems are very low, unlike MPL.

D. Software Product Line Architecture Evolution

Thiel and Hein [21] propose product lines as an approach to automotive system development because product lines facilitate the reuse of core assets. The approach of Thiel and Hein enables the modeling of product line variability and describes how to manage variability throughout core asset development. Furthermore, they sketch the interaction between the feature and architecture models to utilize variability.

Holdschick [22] addresses the challenges in the evolution of model-based software product lines in the automotive domain. The author argues that a variant model created initially quickly becomes obsolete because of the permanent evolution

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-532-6

of software functionalities in the automotive area. Thus, Holdschick proposes a concept how to handle evolution in variant-rich model-based software systems. The approach provides an overview of which changes relevant to variability could occur in the functional model and where the challenges are when reproducing them in the variant model.

Automotive manufacturers have to cope with the erosion of their ECU software. The work in [2] proposes a systematic approach for managed and continuous evolution of dependable automotive software systems. It is described how complexity of automotive software systems can be managed by creating modular and stable architectures based on well-defined requirements. Both architecture and requirements have to be managed in relation. Furthermore, to face the lack of flexibility of existing hieratic automotive software systems development approaches, they are focusing on four driving factors: systems engineering and agile function development, feature and function driven team development, agile management principles, and a seamless tooling infrastructure supporting continuously and iteratively evolving automotive software systems in a flexible manner.

To counteract erosion it is necessary to keep software components modular. But modularity is also a necessary attribute for reuse. Several approaches deal with the topic reuse of software components in the development of automotive products [1][23]. In [1], a framework is proposed, which focuses on modularization and management of a function repository. Another practical experience describes the introduction of a product line for a gasoline system from scratch [23]. However, in both approaches a long-term minimization of erosion is not considered.

A previous version of our approach is described in [3] focusing on the key ideas of the management cycle for product line architecture evolution. Furthermore, an approach for repairing an eroded software consisting of a set of product architectures by applying strategies for recovery and discovery of the product line architecture is proposed.

III. Overall Development Cycle

Our methodology for managed evolution of automotive software product line architectures is depicted in Figure 1. The methodology consists of two levels of development: The cycle on the top of Figure 1 constitutes the development activities for product line development, whereas the second cycle is required for product specific development. Not only both levels of development are executed in parallel but even the activities within a cycle may be performed 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 subsections, we will explain the basic activities of our approach in detail by referring to the terms depicted in Figure 1. Table I gives a brief overview on the objectives of each of the 12 activities, including inputs and outputs.

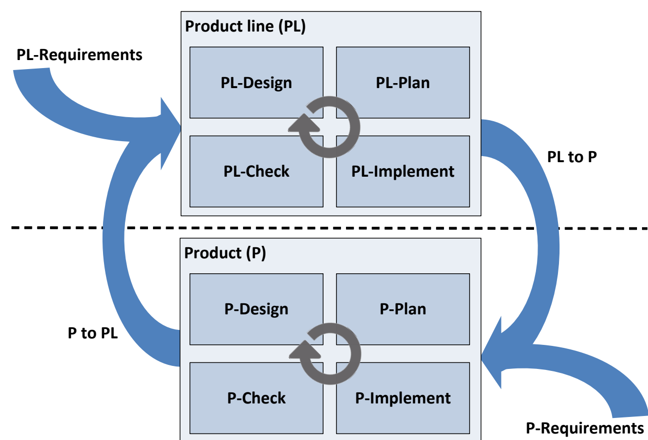

Figure 1. Overall development approach

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.

A. *Planning and Evolving 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 [24]. 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 [25], e.g., activity diagrams are considered for the validation of system requirements by directly executable models including an approach for symbolic execution and thus enabling validation of several products simultaneously.

(`P to PL`) Artifacts of the developed product from the product cycle in Figure 1 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 [2], we propose methods and techniques for a good architecture

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-532-6

TABLE I. EXPLANATION OF THE ACTIVITIES IN FIGURE 1.

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

design. Based on these methods and techniques a new PLA is defined (called PLA vision) taking the new requirements (`PL-Requirements`) and product related information (`P to PL`) into account. To assess the quality of the designed PLA, it is necessary to measure complexity and to describe the results numerically. In particular, we consider properties such as cohesion, coupling, reusability and variability in order to draw conclusions about the quality of the PLA.

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

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

B. Automotive Product Development and Prototyping based on Software Product Lines

(`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, 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 [26], 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 [27]. Then the implementation is improved to pass the test case. Based on the approach in [26] we use the tool Time Partition Testing (TPT) because it suits particularly well due to the ability to describe continuous behavior [28]. 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. In contrast, the product line cycle in Figure 1 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

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-532-6

planned elements of an iteration are implemented according to the test-driven approach of [26].

C. Architecture Conformance Checking for Automotive Software Product Line Development

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

Each planned product refers to a set of implementation artifacts, called modules. These modules constitute the product architecture. The aim of `PL-Check` and `P-Check` is the minimization of product architecture erosion. In [12], 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 [12], the rules are expressed as logical formulas which can be evaluated automatically to the compliance to the PLA.

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

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

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

IV. Architecture Description Language

Evolution of the logical architecture and module architecture in product line and in product development involves a huge number of architecture elements and their relations. To handle this complexity, model based techniques are used within our methodology.

To develop the logical architecture and the module architecture using model based techniques we defined a description language by a metamodel. For each activity in our approach instances of the metamodel with several *views* are modeled. Each view focuses on different architecture elements. The product line phase deals with architecture elements for several product architectures. To derive product architectures from the product line phase *variant handling* has to be considered. We will describe these concepts of our approach in detail in the following.

EMAB metamodel: The **E**inheitliche **M**odulare **A**rchitektur **B**eschreibung (EMAB) is used to describe the decomposition structure and connection structure of logical architectures and module architectures.

Figure 2 depicts a simplified part of the metamodel that shows the abstract syntax of the two architecture layers DESIGN and IMPLEMENT. The architecture elements of these two layers are used to model product lines and product architectures. In the following, we describe the two layers in detail.

The DESIGN layer contains architecture elements to describe abstract software aspects. `LogicalArchitectureElement` is used to decompose these aspects into groups of corresponding implementation artifacts. Some of those elements may have dependencies with other elements of the logical architecture. In this case `LogicalElementConnection` connects exactly two logical architecture elements as a directed connection between one source and one target element. Each `LogicalArchitectureElement` can be referenced by a number of connections. The connection between two elements semantically allows the communication constrained by the source/target direction.

In the IMPLEMENT layer, code relevant software aspects are described. Thus, `ModuleArchitectureElement` decomposes the software code aspects into groups. For example, a header file and a c-code file of a certain software application are represented by a `ModuleArchitectureElement`. Directed dependencies between

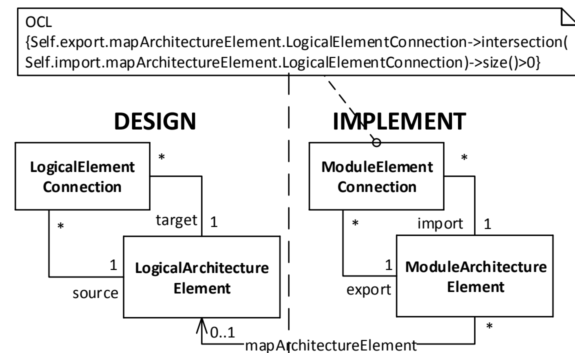

Figure 2. Simplified EMAB metamodel

exactly one export and one import element are connected by the `ModuleElementConnection`. Each `ModuleArchitectureElement` can be referenced by a number of connections. The connection between two elements semantically allows the communication constrained by the import/export direction.

`LogicalArchitectureElements` have to be referred to implementation artifacts for product development. Therefore, the EMAB metamodel determines that the `ModuleArchitectureElement` can reference at most one `LogicalArchitectureElement` using the `mapArchitectureElement` to determine the appropriate module elements of a logical element.

Connections between two `LogicalArchitectureElements` have to be properly considered at the implementation in the IMPLEMENT layer: Connections between `ModuleArchitectureElements` have to be conform with the connections specified in the DESIGN layer. To ensure that connections are realized properly, conformance rules are applied. One example OCL rule is shown in Figure 2 that constraints the `ModuleElementConnection` as context element for the check.

Views: The DESIGN layer focuses on the logical architecture. Figure 3 represents the DESIGN view as block diagram with two instances of logical architecture elements and the connection between them. The roles `source` and `target` indicate the direction of the connection.

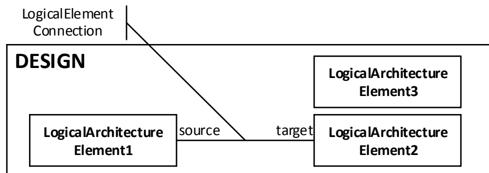

Figure 3. DESIGN view as part of the metamodel instance

The IMPLEMENT view in Figure 4 represents the module architecture instances as blocks and their connection as connection instances. Moreover, each module architecture element is referencing one logical architecture element represented by dashed connections between a module element block and logical element block.

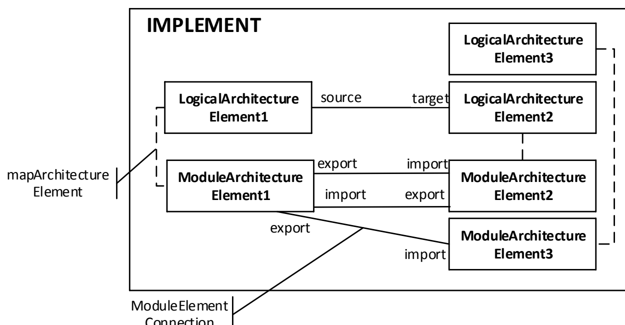

Figure 4. IMPLEMENT view as part of the metamodel instance

Figure 5 shows an example for the CHECK view, checking the conformance rule on connections of the DESIGN layer's logical architecture elements and IMPLEMENT layer's module architecture elements. The check of the OCL rule in the middle of Figure 5 is fulfilled as the specified connection between the two module architectures elements correspond to the connection between the mapped logical architecture elements. However, the further checks of the OCL rule fail: In the first case, the direction from import to export constitutes a violation. And in the next case, as `ModuleArchitectureElement3` maps to `LogicalArchitectureElement3` no connection is allowed from `ModuleArchitectureElement1`.

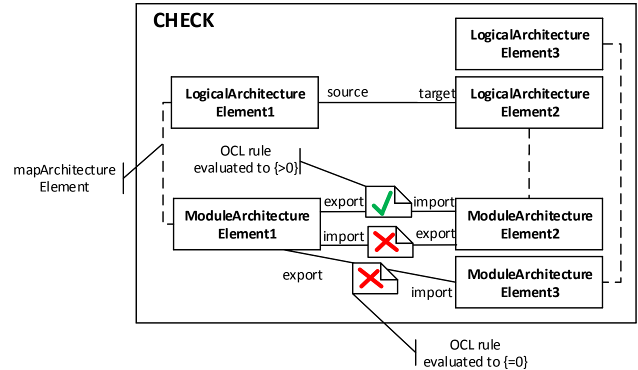

Figure 5. CHECK view as part of the metamodel instance

Variant handling: In product line development, architecture elements of module and logical architectures are realized to be reused in several software products. The architecture models enclose decomposition variants and behavior variants. During product development, the decomposition variants and the behavior variants have to be determined. Therefore, a further part of the EMAB metamodel provides the syntax to describe structure variants, behavior variants and valid selection conditions. Selection conditions are necessary to derive architectures and to derive behavior for product development. The part of the EMAB metamodel dealing with variants is out of the scope of this contribution in order to focus on the methodology.

Version handling: Each architecture element of the product line development and of the product development is kept in a repository. The repository provides a version control capability. A modified or created element is committed with a unique version ID into the repository. Predecessor relations are defined in case of modifications of an existing version. The repository also enables the selection of elements for product line development or product development. The part of the EMAB metamodel dealing with versioning is out of the scope of this contribution in order to focus on the methodology.

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

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

A. System Structure and Context of BSU

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

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

The software system realizes a set of feedback controllers to reduce the disturbances caused by other systems or to switch on the vacuum pump, respectively. Since it makes no sense to use all controllers at the same time it is necessary to coordinate their activation. Besides the controlling of BSU vacuum and the coordination of controllers, the software has to provide valid pressure information all the time. In order to realize that the software selects from several sensors the one that provides the best quality of pressure information. The logical view of the designed architecture is presented in Figure 6.

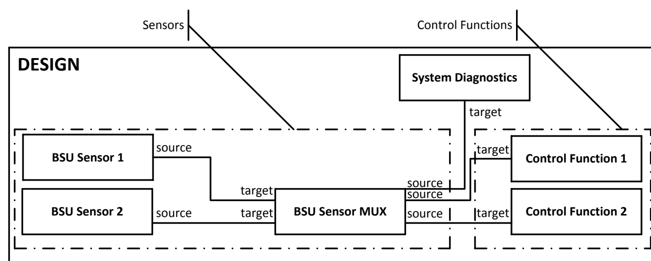

Figure 6. Logical view of the software architecture of BSU

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

The `control functions` component is decomposed using a similar technique. Every control function is realized by a specific component. These components provide standardized interfaces for communication with subsequent vehicle functions, which must follow the BSU commands, e.g., disable the start-stop system (not depicted in Figure 6).

B. Evolution of BSU

As it was customary in the automotive domain, BSU's hardware and software have been implemented by various suppliers in the past. The requirements for the functionalities of the system were the same for all suppliers, but there were differences in the type of implementation by the respective suppliers. During the further development of the system over many years, new requirements had to be continuously implemented. Examples of this are the support of various engine variants such as otto, diesel and electric engines. As the range of functions increased, the essential complexity grew; however, the accidental complexity [29] has increased disproportionately. The growth of accidental complexity results from a "bad" architecture with strong coupling and a low cohesion which have evolved over the time. Despite extensive further development of the system, the original structure of the software was not adequately adapted. Overall, the monolithic structure of the software remained. The software consisted of a single software module, which, however, was internally characterized by increasing accidental complexity. The variability was realized completely by annotations. Thereby, the system's maintainability and expandability has been complicated additionally.

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

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

C. Application of our Approach to BSU Further Development

In this section, we will outline the evolution of BSU further development, described in the previous section, mapped to the overall development cycle visualized in Figure 1. As mentioned in Section V-B, the development started in 2012 and continues until today. We will pick out the milestones of this evolution process and explain in detail, how our approach supports the management of development. Therefore, we will

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-532-6

describe the further development of the BSU chronologically. The architecture of the BSU at this point is equivalent to Figure 6.

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

After assessing and determining the new PLA vision, the `PL-Plan` activity starts. It was decided to realize the new PLA vision in two iterations, per iteration cycle, one of the new components should be implemented completely. Regarding to the development plan in activity `PL-Implement` the first component was implemented. `PL-Check` activity is triggered after the new component is fully implemented. In this activity, the conformance of the implementation is checked against the planned architecture (PLA vision), as illustrated in Section IV. The outcome of the checks was positive so the next iteration was started.

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

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

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

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

After evaluating the product realization all adaptations and changes of architecture and implementation are forwarded to the product line architecture level by a `P to PL` activity. These are inputs for the next `PL-Design` activity, thereby it had to be decided which changes should be integrated into the product line architecture and its implementation or otherwise which had to be declared as a "special" solution. In our case the coordinator concept was established in the product line. The final architecture is visualized in Figure 7 including all newly developed control functions, the coordinator component, and the additional connections between the control functions and the coordinator for the controlling of activation.

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

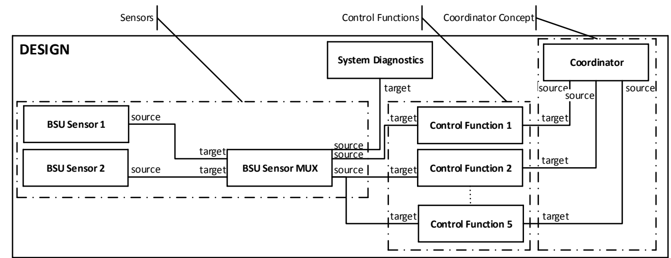

Figure 7. Logical view of the software architecture of BSU including the coordinator concept and the three new Control Functions

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

VI. 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 the following, we focus on the applicability of the product line and product development activities. Two criteria are important to evaluate. First, the amount and kinds of modifications on architecture elements calling this *complexity controlling*. Second, the amount and kinds of design configurations calling this *variant controlling*.

Table II shows the result of the quantitative analysis. The data record for the quantitative analysis refers to the development of the BSU software and the product realizations consisting of the BSU software and further vehicle functions. The record contains the version control graph of the past 5 years of BSU software development, called *repository* in the following. Each node is a version of an architecture element or realized

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

	Count	*Number of versions*	*Average number of versions*	*Min. number of versions*	*Max. number of versions*
LAE	15	15	15/15 = 1	1	1
MAE	15	58	58/15 ≈ 4	1	6
Projects	21	146	146/21 ≈ 7	1	12

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

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

product. Edges connect two subsequent versions. Table II shows the number of logical architecture element (LAE) versions, of module architecture element (MAE) versions, and of project versions from the record. Modifications were triggered by the realization of BSU software `PL-Requirements` or by the realization of products due to `P-Requirements`.

Table II shows the count of 15 LAE referring to modifications at the DESIGN layer and the count of 15 MAE referring to modifications at the IMPLEMENT layer. The kind of modifications refers to the connection structure and to the architecture element structure of the appropriate MAE. Each LAE is available in exactly one version in the repository. Thereby, the current state of the logical architecture is represented which is unmodified since the beginning of the record. Unfortunately, the data of prior development stages of the BSU software logical architecture is not considered by the record due to data protection reasons. In total, 58 versions for MAE exist. A module element of the module architecture was modified in minimum 1 time, in maximum 6 times, and in average 4 times. Thereby, each version of the MAE is mapped in this case to exactly one version of the appropriate LAE.

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

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

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

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

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

Variant controlling: The term variant in the case of BSU software describes a software architecture variant reused to realize a software product. Thereby, each project version refers to exactly one design configuration to define architecture elements for reuse that are contained in the software architecture variant. Modifications of the logical and module architecture can introduce violations on expected derivable structure variants. To handle such violations the control of variants must be applied to the modifications. The control of such architecture rule violations is applied during the `Check` activity of the product line development considering the versions corresponding to the IMPLEMENT layer and to the DESIGN layer. After applying our methodology, no violations are found in the past 5 years of development. This corresponds to the result of complexity evaluation where conformance of the EMAB layers is confirmed.

VII. CONCLUSION AND FUTURE WORK

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

We demonstrated our methodology on a real world case study, a brake servo unit (BSU) software system from automotive software engineering. As a result, we could avoid architecture erosion for several years. All 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.

As a future work, we aim at realizing a tool-chain which enables the architecture description of the different architectures (PLA, PA, including versioning), the measure and evaluation of quality attributes, as well as the integration of the ArCh-Framework [12]. 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 which enables the description and visualization of variability.

REFERENCES

[1] 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, ser. EMSOFT'04. ACM, 2004, pp. 203–210.

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

[3] B. Cool et al., "From Product Architectures to a Managed Automotive Software Product Line Architecture," in Proceedings of the 31st Annual ACM Symposium on Applied Computing, ser. SAC'16. ACM, 2016, pp. 1350–1353.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] S. Herold and A. Rausch, "Complementing model-driven development for the detection of software architecture erosion," in Proceedings of the 5th International Workshop on Modeling in Software Engineering, ser. MiSE '13. IEEE Press, 2013, pp. 24–30.

[13] I. John and J. Dörr, "Elicitation of Requirements from User Documentation," in Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03), ser. Essener Informatik Beiträge, vol. 8. Essen: Universität Duisburg-Essen, 2003, pp. 3–12.

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

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

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

[17] 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.

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

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

[20] 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.

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

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

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

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

[25] 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.

[26] H. Peters et al., "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, 2014, pp. 294–301.

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

[28] E. Lehmann, "Time Partition Testing," Ph.D. dissertation, Fakultät IV – Elektrotechnik und Informatik, TU Berlin, 2004.

[29] F. P. Brooks, Jr., "No Silver Bullet Essence and Accidents of Software Engineering," Computer, vol. 20, no. 4, Apr. 1987, pp. 10–19.