Mastering Erosion of Software Architecture in Automotive Software Product Lines

Arthur Strasser[1], Benjamin Cool[1], Christoph Gernert[1], Christoph Knieke[1], Marco Körner[1], Dirk Niebuhr[1], Henrik Peters[1], Andreas Rausch[1], Oliver Brox[2], Stefanie Jauns-Seyfried[2], Hanno Jelden[2], Stefan Klie[2], and Michael Krämer[2]

[1] TU Clausthal, Department of Computer Science, Software Systems Engineering
Julius-Albert-Straße 4, D-38678 Clausthal-Zellerfeld, Germany
[2] Volkswagen AG, Powertrain Electronics
P.O. 16870, D-38436 Wolfsburg, Germany

Abstract. Most automobile manufacturers maintain many vehicle types to keep a successful position on the market. Through the further development all vehicle types gain a diverse amount of new functionality. Additional features have to be supported by the car's software. For time efficient accomplishment, usually the existing electronic control unit (ECU) code is extended.
In the majority of cases this evolutionary development process is accompanied by a constant decay of the software architecture. This effect known as software erosion leads to an increasing deviation from the requirements specifications. To counteract the erosion it is necessary to continuously restore the architecture in respect of the specification.
Automobile manufacturers cope with the erosion of their ECU software with varying degree of success. Successfully we 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 were extracted by explicit projection of the architecture variability and decomposition of the original architecture. After initial application, this approach was capable to restore the BSU architecture recurrently.

Keywords: Architecture design, Reuse, Engineering methodologies, Model driven development, Software product lines, Software erosion, Automotive

1 Introduction

In the automotive sector, global markets have to be served, in which different requirements for the vehicle exist, e.g. country and culturally specific. Thus, specific adjusted variants of the vehicle types have to be developed and produced. Due to high cost pressure variants and vehicle types can not be developed independently. Instead, potential synergies have to be exploited.

As a result, components are built in multiple vehicle variants with different configurations. Hence, this approach also applies for the corresponding software modules.

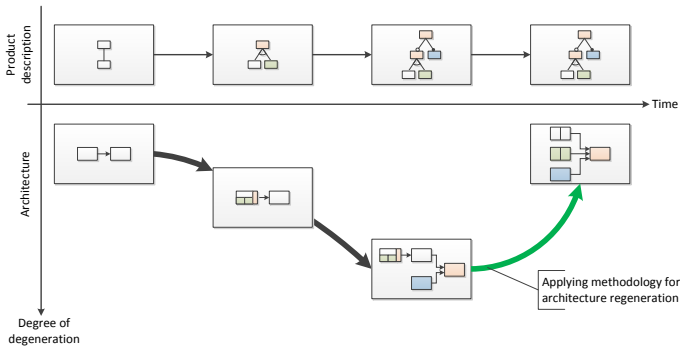

Fig. 1. Degeneration of the architecture based on iterative software development

The context, e.g. the available sensors and actuators, in which the software module is used, may differ between vehicle variants. Consequently, software modules have to be able to deal with this variability. For this, product line approaches have proven to be successful in the past.

As shown in Fig. 1, a product line is designed initially and developed over time. It makes no difference whether the product line is explicitly planned or exists only implicitly in the minds of the participants. This product line with its variance regarding the available sensors, actuators and software features provides the basis for the development of an appropriate architecture.

Over the lifetime of the product line new variation points and variants are added. For logical reasons functioning systems are not developed from scratch, but the current state of their architecture and implementation is consequently developed evolutionarily. Since the implementation of new features has to be as flawless, cost-effective and timely as possible, this type of development is often applied in cases where the risks for faults and an explosion of costs and time are the lowest. Apparently that is not always the ideal place in regard to the quality of the resulting software, thus it leads to the erosion of the originally planned architecture.

At some point structural difference between the product line and the architecture size reaches a level that it will be increasingly difficult to integrate new features into the system with the required quality, cost and time. It is then necessary to fundamentally rethink and subsequently renew the architecture. At present time architecture renewing is a highly unsystematic and creative process. Using an example from practice we show how a fundamental procedure may be designed to optain a renewed architecture in a methodical and strutured fashion, starting from a degenerated architecture.

[5] shows an approach to extract a product line from a user documentation. Our approach has a similar objective, but assumes models in form of block diagrams.

This paper is organized as follows: Section 2 gives an overview of the state of research in the topic of our pursued approach. The case study is introduced

in Section 2. Section 3 outlines the main steps for the extraction of product lines from the existing variants of the brake servo unit. Section 4 summarizes the results of the proposed approach and provides an outlook on further work.

2 Background

In the iterative development process new software product variants of high quality are developed on existing software artifacts. Initial design concepts serve reuse and further development of product variants in each cycle of development. If the requirements specification no longer corresponds to the architecture and the implemented software products, the consistency between those artifacts must be restored.

The aim of our approach is the approximation between the architecture and the specification to minimize the so-called software erosion. In Fig. 2 this step is shown as activity for the architecture regeneration. Then the result artifacts can be reused and further developed in the phase of *long-term evolution*.

2.1 Structure of ECU Software

In the automotive scope ECU software is usually divided into application software and basic software. These parts are well known as abtractions layers from OSEK-COM layered model [9]. The application software consists of various components, which are the building blocks of the system's architecture and are derived from the requirements. The components are more or less distinctly linked to the precedent requirements of the system. The basic software is composed of the ECU operating system and hardware-dependent functions, both provided by

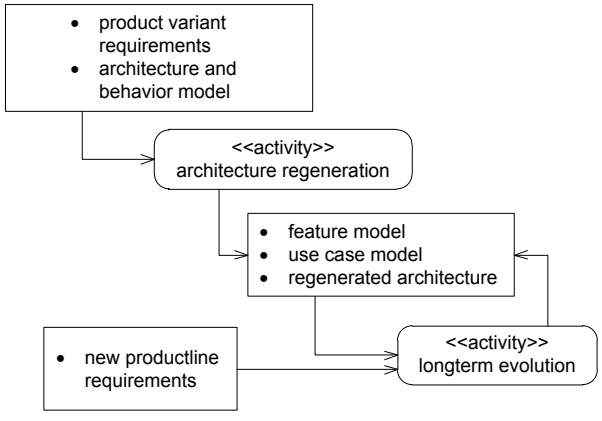

Fig. 2. Required activities for mastering erosion in software product line

the third-party supplier [6]. The OEM and supplier software interact through standardized software-interfaces like AUTOSAR or ASAM-MDX. It is assumed that both parts of the basic software work efficient and without flaws, which affect the application software.

After the initial implementation of a component, various changes within the component's lifespan can occur. The existing components will be reused or modified, every time the requirements of the actual system or a related system change. Similar to a black box approach, the modifications of the affected components are often only documented through the alteration of their interface [10].

Changes of the BSU software was documented in the same fashion, leading to a constant reduction of the cohesion regarding the initial purpose that is defined by the component's requirements.

2.2 Erosion

The challenge in the development of software architecture is to minimize the effort for implementation of design changes, particularly if the architecture yet has been further developed. The fundamental problem is that all design decisions based on the requirements during the domain analysis are only implicitly included in the evolving architecture. If this information is not available while changing the existing architecture, these changes can violate initial design decisions. This effect is called *knowledge vaporization* and conclusively leads to architecture erosion [11, 1].

The implemented architecture is based on the requirements specification, resulting in a component-based architecture description [9]. This software architecture description provides the basis for the reuse of existing software components and their iterative development through the implementation of new ECU software functions.

The software components realize requirements specified by the stakeholders depending on the vehicle model and its installed hardware such as sensors or actuators. In the first step the definition of the component interface is carried out followed by the component definition itself [9]. The resulting artifacts are the interfaces and the behavior model which are managed with tool support. Implemented variances which arise from the vehicle model, functional specifications and the installed hardware are also specified in the result artifacts. By querying a condition of the behavior model e.g. the presence of a vehicle-specific sensor can be specified. The interfaces can be parameterized in this manner as well. This increases the complexity of the software component.

This approach for the software component implementing complicates component reuse and development for new and existing vehicle models. New dependencies arise by the specification of variances through conditions in several functionally distinct parts of the behavior model. If this kind of variant specification is implemented repeatedly, the behavioral model parts, originally specified as independent, can be difficult to identify. If the conditions also depend upon each other, the distinction between a valid and invalid variance gets complicated

without initial design knowledge. This design freedom does not have to be documented explicitly [12]. The risk of architectural erosion increases. In order to ensure maintainability and extensibility, it is pursued to keep this risk in the long term to a manageable level.

2.3 Software Product Line Extraction

The aim of extraction is to identify all the valid points of variation and the associated functional requirements of block diagrams. The block diagrams represent the behavior model and the interfaces of the software component.

The development of software product line engineering consists of two essential steps:

- domain engineering (core asset development)
- application engineering (product development from core assets)

By applying the software product line approach new software products are developed from reusable core assets. For specification of all product line properties two analysis techniques are suggested in SEI [2]. In our case, the model-based analysis method is used to specify product line variability from existing models.

The PLUS approach permits variability analysis based on use case scenarios and the specification of variable properties in the FODA feature model. To carry out the variability analysis, we use the PLUS approach [4] to describe the variability model.

The use cases of the product line core are central components of the product line and thus present in every derived product. Optional use cases describe tasks that do not have to be carried out by all products. These may or may not have to be carried out by a particular product. On the other hand alternative use cases mutually exclude each other, if they are located within the same group. This means that two alternative use cases can not simultaneously be part of a derived product. After the identification of all core and variable use cases these are grouped into features. Commonly reused scenarios are each assigned to a feature. This assignment may be stated in a table.

In conclusion the dependencies between the features in the variability model are specified according to FODA. Thereby the root node always groups all applications defined as the core scenarios. All remaining features, such as the use cases, carry the attributes *optional* or *alternative* and can also exhibit dependency relationships defined through PLUS. A validly selected feature set characterizes the set of requirements of a particular software product.

Moreover a requirement specification is developed using use case scenarios and the variability model that describes all the product line requirements. The requirements specification contains the coarse architecture based on the use case and variability model providing the starting point for the product line re-engineering.

The functional requirements and the initial design decisions of the behavior and interface model are extracted with the aid of the result artifacts of the product line extraction.

In addition, this approach allows further development of the software. Dependencies between the variants are recorded in the extraction phase. In this way, the modularization during the software component design is supported in the long term. New variation points must be recorded in the product line artifacts, be associated with the requirements and also provide validation for the subsequent architecture design.

2.4 Software Architecture Re-Engineering

Based on the result artifacts of the production line extraction, the architecture components are designed in the next step. The goal is the fulfillment of the architecture quality requirements, such as modularity and reusability of components. In particular, the quality should be maintained on subsequent architectural changes.

A key component of the approach is the design of reusable software components by the selection of features in the variability model. All valid variants and their requirements can be derived from the variability model. The use cases are associated with the respective features. In this way the variability does not need to be explicitly mapped to the architecture.

Based on the existing architecture and variability model the coarse architecture is refined. For each feature, software components are described, taking into account the original architectural models. The variants were detected in the step of the product line extraction. Based on the features and use cases, components are designed which are either reusable for each product or only in certain products. The extend and include relationships in the use case model characterize variation points for further development. This can be relevant during the component design.

After the design, the components are connected depending on their interfaces. The connection depends on the mapped features and the variability model, resulting in a product line architecture.

3 Experimental Details

The purpose of the brake servo unit (BSU) in cars is to assist the driver by boosting his brake force. To achieve this, a vacuum (for example from the intake manifold) is used to achieve a pressure difference on two sides of a diaphragm. The pressure difference results in an additional force inside the braking system.

The software's objective is to guarantee a sufficiently high pressure difference in order to achieve the required boost. Initially, only one sensor was available for pressure measurement. The development of the controller fulfilling the requirements (e.g. by adjusting the engine speed) was model-based using block diagrams.

Over time, new systems were added in cars, to which the BSU software had to be adapted. This concerned, for example, new sensors available after the introduction of electronic stability control systems (ESC), as well as new

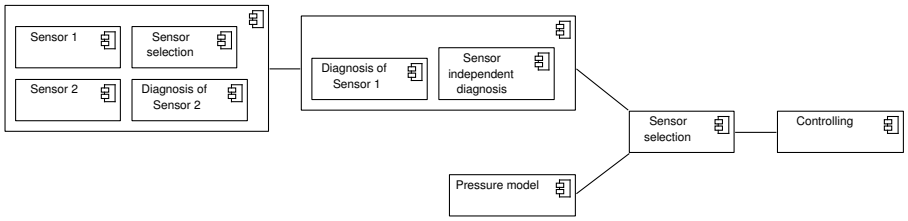

Fig. 3. Component diagram including the sensors and actors of the BSU

actuators (e.g. an electrical vacuum pump). In addition cross effects of new functions such as *Start-Stop* caused disturbances which the existing controllers could not handle.

As shown in Fig. 3 the continuous erosion of the software architecture led to wider component interfaces and a reduction of the components' cohesion.

We were able to identify two major variation points within the BSU software: *sensors* and methods of *vacuum generation*. After examination of the models, we found two versions of the variation point "methods of vacuum generation", specifically "intake manifold evacuation", and "vacuum-pump evacuation".

The calculation of the pressure model, selection of the sensor signal, pressure controller, and actuator controller are located in the kernel of the BSU software. The use cases "sensor 1" and "sensor n" capture data from differing sensors and extend the core use case "Choose sensor value". Each pressure control method is encapsulated in a separate use case. They extend the kernel use case "Control pressure difference".

For a more detailed description of the use cases, templates [8] are used to document the variability adequately.

We were able to decompose the function into smaller components with higher cohesion with respect to these use cases (Fig. 5) and the feature model (Fig. 4) and developed an architecture template model (Fig. 6) to describe the possible

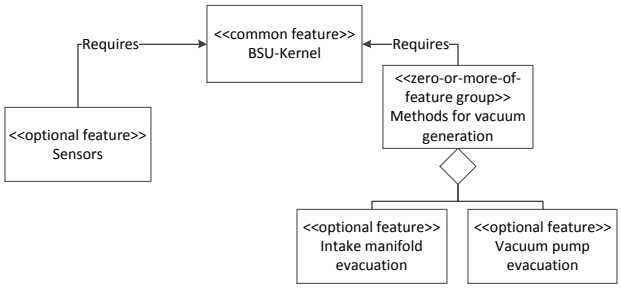

Fig. 4. Resulting variability model after the architecture regeneration activity

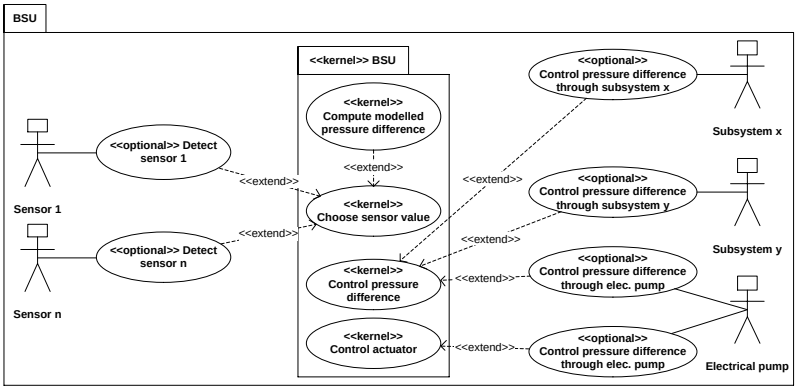

Fig. 5. Use case diagram of the BSU. Sensors and subsystems are displayed as UML actors.

connections of the components. This model can be used to derive an architecture for every configuration.

4 Results and Discussion

For an over the years iteratively developed software product line as the brake servo unit, it is found that the software architecture is getting worse over time, as it exhibit weak cohesion, for example.

In the case of the examined BSU software, we concluded that no explicit definition of the product line was done. The product line existed only in the minds of developers. Originally there was a general variant of the software, which evacuated 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

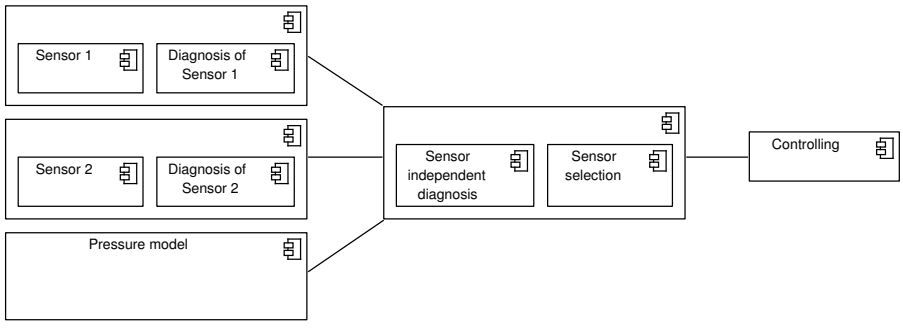

Fig. 6. Resulting domain specific template model of the BSU software

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 difficult to overlooking, 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. For this purpose, we have developed a methodology consisting of three successive steps *variant analysis*, *requirements specification* and *product line re-engineering*. A brief summary of the methodology is illustrated in Fig. 7. In the following subsections we explain each step on the example of the BSU software.

The usage of software product lines is an established method in the field of software engineering. An alternative to SPL would be an opportunistic concept, where for each product the whole set of functions is available for reuse [7]. Because of the limited amount of ECU memory this concept is not an option.

Feature-Oriented Domain Analysis (FODA) is a feasible approach for variability analysis in complex systems [3]. It offers an clear overview of the components' variability. On the other hand feature diagrams only provide a loose connection to the specified functionality of the particular components in a system. To extract variably in direct relation to the system functionality our approach is based on use cases, similar to PLUS [4]. For complex systems like the BSU, the

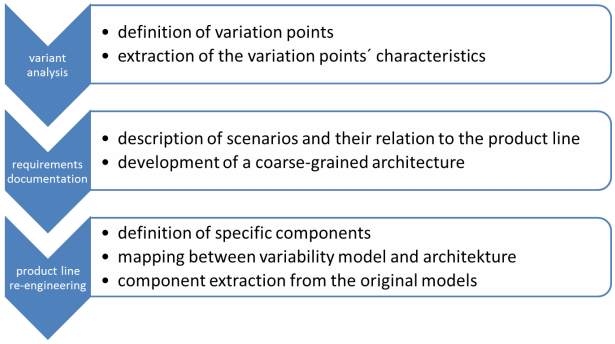

Fig. 7. Procedure of the architecture regeneration approach

inference to separate use cases involves an increased effort than the inference to feature models.

5 Conclusion

We proposed an approach to support effectively avoiding erosion of software architecture by a product line which is enhanced iteratively. By continuously applying architecture extraction and regeneration, a strong cohesion of the software components is achieved improving maintainability and extensibility of the software.

The software components are one-to-one mapped to variation point features. Every component is reusable by selecting variability model features nodes. Each encapsulates a clearly defined requirement set of sensor, actuator and control engineering functionality, which does not infringe the single responsibility principle. All components have generalized interfaces. This enables to interconnect all components according to feature selection. We call the final design "template architecture".

One important advantage of the template architecture is modularity. Evolution can be done on every single component without considering further BSU component dependencies. New functional requirements must be realized with respect to the existing use case model, variability model and template architecture, following our two staged methodology. This minimizes the risk to violate the requirements specification by making further design modifications. In addition, the effort for software configuration in certain hardware environments is decreased. By realizing variability on the level of software architecture, readability of all model components is improved.

Initially the BSU software was developed ECU-independent. The corresponding behavior-model does not include any memory or processor specific data types but was configurable by parameters depending on used actuator- and sensor-hardware. We recommend to focus on those features in automotive software architectures when applying our methodology.

Our approach was demonstrated by a real-world case study of a brake servo unit. The results of the case study are relevant for further automotive systems, too. The case study shows how the methodology is used to design modular and reusable software components, focusing on sensor and actuator hardware components and characterizing the variability of the car equipment configuration.

But there are still questions about handling variability in the architecture design phase e.g. introducing a new feature, a relation between features or generalize component interfaces. The degree of freedom in making not documentable architectural design decision using our methodology is still high. Moreover there is also a tradeoff between the amount of used microcontroller resources and modularity in the architecture.

The BSU software based on the extracted product lines has already been partially implemented. Experiences concerning extendability of the software could

be gained. Future work can refer to elaborating an integrated tool-chain supporting the modeling languages used in our approach adequately.

References

1. Bosch, J.: Software architecture: The next step. In: First European Workshop on Software Architecture. EWSA '04. St Andrews, Scotland, UK. (2004)
2. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley (2001)
3. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wäsowski, A.: Cool features and tough decisions: a comparison of variability modeling approaches. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems. VaMoS '12, New York, NY, USA, ACM (2012) 173–182
4. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional (2004)
5. John, I., Dörr, J.: Elicitation of requirements from user documentation. In: Ninth International Workshop on Requirements Engineering: Foundation for Software Quality. REFSQ '03. Klagenfurt/Velden. (2003)
6. Kindel, O., Friedrich, M.: Softwareentwicklung mit AUTOSAR®: Grundlagen, Engineering, Management für die Praxis. Dpunkt.Verlag GmbH (2009)
7. Krueger, C.W.: Introduction to the emerging practice software product line development. Methods & Tools **Volume 14** (2006) 3–15
8. Lipka, C.: Architekturoptimierung fr Steuergerte-Software am Beispiel eines Bremskraftverstrkers aus dem Automobilbereich. Masterarbeit, TU Braunschweig, Institut für Programmierung und Reaktive Systeme (2012)
9. Schäuffele, J., Zurawka, T.: Automotive software engineering: principles, processes, methods, and tools. Vieweg+Teubner (2010)
10. Schäuffele, J., Zurawka, T.: Automotive Software Engineering. 4.,überarbeitete und erweiterte auflage edn. Vieweg+Teubner (2010)
11. van Gurp, J., Bosch, J.: Design erosion: Problems & causes. Journal of Systems and Software **Volume 61** (2002) 105–119
12. Weber, M., Weisbrod, J., DaimlerChrysler-Research: Requirements engineering in automotive development: Experiences and challenges. In: Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on. (2002)