Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Feature dependencies in automotive software systems: Extent, awareness, and refactoring

Andreas Vogelsang

Technische Universität Berlin, Germany

ARTICLE INFO

Article history:
Received 21 March 2019
Revised 4 September 2019
Accepted 1 November 2019
Available online 7 November 2019

Keywords:
Feature interaction
Automotive
Requirements engineering
Software architecture
Technical debt
Empirical study

ABSTRACT

Many automotive companies consider their software development process to be feature-oriented. In the past, features were regarded as isolated system parts developed and tested by developers from different departments. However, in modern vehicles, features are more and more connected and their behavior depends on each other in many situations. In this article, we describe how feature-oriented software development is conducted in automotive companies and which challenges arise from that. We present an empirical analysis of feature dependencies in three real-world automotive systems. The analysis shows that features in modern vehicles are highly interdependent. Furthermore, the study reveals that developers are not aware of these dependencies in most cases. For the three examined cases, we show that less than 12% of the components in the system architecture are responsible for more than 90% of the feature dependencies. Finally, we propose a refactoring approach for implicit communal components, which makes them explicit by moving them to a dedicated platform component layer.

1. Introduction

Software development in automotive companies is strongly influenced by existing legacy systems, organizational constraints, and complex OEM/supplier relationships (Broy, 2006). Nevertheless, automotive companies are forced to quickly deliver increasingly complex software to keep up with their competitors and other digital products with shorter development life-cycles. In this context, like in many others, short-term goals, such as the delivery of a feature, frequently trump long-term objectives like maintainability or extensibility (Martini et al., 2014).

The development of the software system in an automobile is characterized by a decomposition into vehicle domains such as powertrain, body, chassis, driver assistance, and infotainment. Within these vehicle domains, subsystems group and structure several vehicle features that provide functionality to the driver or other external systems (Broy et al., 2007). Examples for vehicle features are *airbag, cruise control*, or *start-stop system*.

Automotive companies try to keep features as independent as possible from each other because they usually structure their organization and resources based on features (e.g., *airbag* and *cruise control* can be developed in completely different departments). However, in the past years, the different features of a vehicle got more and more interconnected to provide innovative behavior (Broy, 2006). For example, the central locking system integrates the pure functionality of locking and unlocking car doors with comfort features (such as adjusting seats, mirrors, and radio tuners according to the specific key used during unlocking), with safety/security features (such as locking the car beyond a minimum speed, arming a security device when the car is locked, and unlocking the car in case of a crash), and with human-machine-interface features, such as signaling the locking and unlocking using the car's interior and exterior lighting system.

A feature is implemented through a network of communicating components. Technically, a component is a piece of software deployed to a hardware execution unit, which is connected to one or more bus systems that provide signals from all kinds of other components. The signals on a bus system are available to all components connected to that bus. Therefore, it is a common practice of developers to (re)use any signal that is available on the bus system to implement or adapt a feature, regardless of the origin of that signal. This practice leads to behavioral dependencies between features, some of which are intended and some of which are unintended.

Behavioral dependencies between features (a.k.a. feature interactions (Zave, 1999)) have been observed and addressed first in telecommunication systems (Calder et al., 2003) followed by studies on Internet applications (Crespo et al., 2007), service systems (Weiss et al., 2005), automotive systems (Vogelsang and Fuhrmann, 2013), software product lines (Jayaraman et al., 2007), computational biology (Donaldson and Calder, 2012), and in many

E-mail address: andreas.vogelsang@tu-berlin.de

https://doi.org/10.1016/j.jss.2019.110458
0164-1212/© 2019 Elsevier Inc. All rights reserved.

other fields outside of computer science. Several studies show that feature dependencies have a negative impact on maintenance efforts (Ribeiro et al., 2011; Cafeo et al., 2016), increase the likelihood of integration failures (Cataldo and Herbsleb, 2011), and prevent modular reasoning (Kästner et al., 2008).

Since the development of automotive systems is structured according to features, our research goal is to analyze the extent and awareness of feature dependencies in practice empirically. We are interested in how many feature dependencies actually exist in real-world automotive systems, whether the developers are aware of these, and whether the dependencies play a role in the way the systems are built.

To answer these questions, we had the chance to examine three automotive software systems from practice. More specifically, each system was characterized by a set of features that it provides, a set of components with interface descriptions that implement the features, and a feature-component mapping that indicates which components contribute to the implementation of which features. Since there was no notion of feature dependencies in the datasets (nor in any other artifact of the company), we developed an algorithm to extract feature dependencies from the component architecture.

With this algorithm, we found numerous feature dependencies that crosscut the whole system. In a follow-up interview study, we found that the respective developers were unaware of almost 50% of the dependencies. Moreover, we were able to show that feature dependencies are not considered systematically when it comes to restructuring the system's architecture although implicit feature dependencies can be considered as technical debt (Vogelsang et al., 2016). Therefore, we propose a dependency-based refactoring approach that suggests shifting components from features to a dedicated platform component layer if they are strongly affected by feature dependencies.

In summary, we describe the following contributions in this paper:

1. We propose an algorithmic approach for extracting feature dependencies from component architectures.
2. By analyzing three automotive software systems from practice, we show that feature dependencies are numerous and crosscut the whole system.
3. By confronting developers with these dependencies and analyzing so-called *service* features, we show that feature dependencies are hardly known and considered in the development of the system's architecture.
4. We propose a dependency-based refactoring approach for system components, which is able to reduce the number of feature dependencies by 90% by refactoring less than 12% of the components in the examined systems.

Structure of the paper: This paper is structured along the questions of extent, awareness, and refactoring of feature dependencies. After providing some background information and introducing the dependency extraction algorithm in Section 2, we analyze the study object systems with respect to extent of feature dependencies (Section 3) and awareness of feature dependencies (Section 4). In Section 5, we introduce our refactoring approach and show its application to the three systems. In Section 7, we present alternative solutions to ours before concluding the paper with a discussion and summary.

Relation to previous work: This article summarizes and extends the work of previous publications (Vogelsang et al., 2016; 2012; Vogelsang and Fuhrmann, 2013). We extend the previous work by the following contributions:

- We extend the analysis of RQs 1–2 and 5–6, which have already been addressed in previous work, by an additional dataset that is larger than the existing two datasets. By this, we enhance the external validity of our previous work. In addition, we provide a more in-depth discussion of the results.
- We extend RQ2 with a new analysis that correlates the number of feature dependencies associated with a component with the position of a that component in a feature processing chain. This analysis shows details about the role of dependencies in different architectural stages of a feature (e.g., sensing, processing, actuation).
- We address a new research question RQ4 in the context of the new dataset. In this RQ, we examine the relation between feature dependencies and so-called service features that developers defined in the new dataset. The purpose of these service features is that they provide platform functionality available for use in other features of the vehicle. The explicit definition of service features in the new dataset allowed us to examine whether feature dependencies are more frequent in service features compared with regular features. This analysis provides an additional viewpoint to the question of how aware developers are of feature dependencies.
- We explain the dependency extraction algorithm in more detail and provide a characterization as pseudo code. In addition, we publish the tool that we developed to perform the feature dependency analysis. This increases the reproducibility and transparency of our analysis and allows other researchers to reuse the analysis.

2. Background

2.1. Features and feature dependencies

The term feature is associated with a great variety of meanings and interpretations in research and industry. Additional terms that are often mentioned in this context are the terms *function* or *service*. Depending on the focus, the term feature may be used to describe distinctive characteristics of a system (Kang et al., 1990; Chen et al., 2005), elements of a functional specification (Shaker et al., 2012; Schätz, 2008), or increments and configuration options in a design or implementation (Liu et al., 2006; Apel et al., 2010).

In this article, we focus on features as elements of a functional specification for a multifunctional system (cf. Broy, 2010; Batory et al., 2004). This means features are used to structure the functionality of a system with the goal to decompose the specification. Decomposition into completely independent features is usually not possible and also not desirable in many cases. The goal is to break down the functionality into features with small and clear interfaces to each other to allow for a modular and distributed development. For our work, it is not important whether a feature also represents a configuration option. Our analysis focuses on features and their dependencies that are part of one specific product.

Based on the different notions of a feature, the notion of feature dependencies also differs. In the context of software product lines, feature dependencies are understood as constraints over the possible configuration space of the product line (Apel et al., 2013a). The constraints may be specified by logic relations between features such as *requires* or *excludes*. We do not focus on this interpretation of feature dependencies in this article. Several researchers focus on code-level implementations of software product lines and the challenges of feature dependencies for the development process. Cafeo et al. define: "In the source code, a feature dependency occurs whenever one or more program elements within the boundaries of a feature depend on elements external to that feature, such as a method defined in one feature and called by another feature" (Cafeo et al., 2016). The effects of such dependencies have been extensively studied in preprocessor-based implementations (Kästner et al., 2008). Ribeiro et al. (2011) and

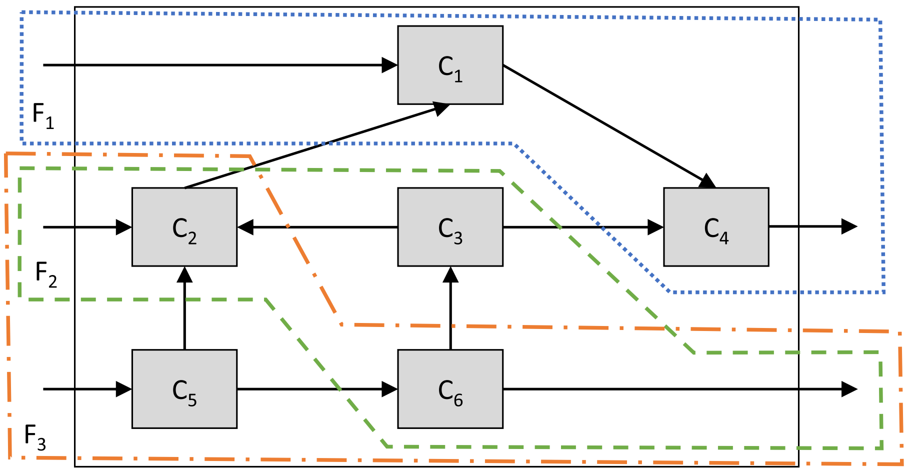


Fig. 1. The components (rectangles) are connected by data channels (black arrows) and form a component architecture of the system (outer rectangle). The vehicle features crosscut this architecture by the set of components that contribute to their implementation (dashed forms).

Cafeo et al. (2016) show that maintenance effort propagates along feature dependencies in code implementations.

Another interpretation of feature dependencies is related to the term feature interaction (Zave, 1999). A feature interaction is "some way in which a feature or features modify or influence another feature in defining overall system behavior." Zave (1999) In most cases, feature interaction is seen as "unwanted" in the sense that the side-by-side integration of two (or more) features into one system leads to system behavior that has not been foreseen and is unwanted in most cases, although all features work as specified. In our work, we focus on this kind of behavioral dependencies between features although we do not care about whether the feature interaction is wanted or unwanted. In fact, for the three cases that we analyze in this article, we assume that most feature interactions contribute to *wanted* behavior because the analyzed systems are in a state where they already have been extensively reviewed and tested. The more interesting question for this article is whether the developers are aware of all the feature interactions and how the interactions are used to shape the system's architecture.

Instances of the feature interaction problem have been observed and addressed first in telecommunication systems (Calder et al., 2003) followed by studies on Internet applications(Crespo et al., 2007), service systems (Weiss et al., 2005), automotive systems (Vogelsang and Fuhrmann, 2013), software product lines (Jayaraman et al., 2007), computational biology (Donaldson and Calder, 2012), and in many other fields outside of computer science.

2.2. Features and component architectures in the automotive domain

As already mentioned in the introduction, automotive software systems are decomposed into vehicle domains, subsystems within these domains, and finally vehicle features within the subsystems. The vehicle features describe coherent behavior that the vehicle provides to its users and external systems. That means a feature describes end-to-end behavior that relates system stimuli to desired system reactions. The implementation of vehicle features is described in terms of communicating components.

Components describe the implementation of vehicle features in a purely logical fashion, i.e., without considering the underlying hardware structure. A network of components describes the processing steps that are necessary to transform the input data into the desired output data. An example of a system that consists of 3 features, which are realized by a network of 6 components, is illustrated in Fig. 1. Components are later implemented as software components, which are executed on electronic computing units (ECUs).

The relation between a feature and a component in the context of this study is the following: A feature is implemented by a set of components that are arranged in a dataflow network. A component can contribute to the implementation of a set of features. Thus, there is an n: m relation between features and components. The set of all components and their connections form a component architecture of the entire system (cf. Broy et al., 2012). The vehicle features crosscut this architecture by the set of components that contribute to their realization (see Fig. 1).

2.3. Feature dependency extraction from component architectures

We want to identify feature dependencies by analyzing the structure of the underlying component architecture and the mapping between features and components. Our initial informal definition states that a feature F_1 depends on another feature F_2 if its behavior is influenced not only by its primary inputs but also by the state or data of F_2. Therefore, if a feature depends on another, there must be some kind of communication relation between the two.

We use the following definition of *feature dependency* within this paper:

Definition 1 (Feature Dependency)**.** A feature *depends* on another feature if at least one component associated with the feature reads a signal that originates from a component associated with the other feature.

In Fig. 1, F_1 depends on F_2 (because C_4 reads a signal that originates from C_3, which is associated with F_2).

Based on this definition of a dependency between vehicle features, we can extract a vehicle feature graph from the component architecture, where each node is a feature and a directed edge indicates a dependency between two features. The resulting vehicle feature graph for the example of Fig. 1 is illustrated in Fig. 2.

The corresponding extraction algorithm is listed in Algorithm 1. The algorithm takes a set of features, a set of components, and a feature-component mapping as input. The feature-component mapping is a total function that maps all features to a non-empty set of components that contribute to the realization of the feature. We assume that all features that are not associated with any component are removed beforehand. The algorithm iterates over all pairs of different features (line 2) to look for dependencies between them. Afterward, the algorithm iterates over the disjoint

Algorithm 1 Feature dependency extraction algorithm.

```
    Input: Set of features $F$,
    Set of components $C$,
    Feature-component mapping: $fcm : F \rightarrow C^+$
    Output: Feature dependency mapping: $fd : F \rightarrow F^*$
1: procedure ExtractFeatureDependencies($F$, $fcm$)
2:     for all $f_1, f_2 \in F : f_1 \neq f_2$ do          ▷ no self-dependencies
3:         for all $c_1 \in fcm(f_1)$, $c_2 \in fcm(f_2)$ do
4:             if $c_1 \notin fcm(f_2) \wedge c_2 \notin fcm(f_1)$ then          ▷ no shared components
5:                 if $signalFlow(c_1, c_2)$ then
6:                     $fd(f_1) \leftarrow fd(f_1) \cup f_2$
       return $fd$
```

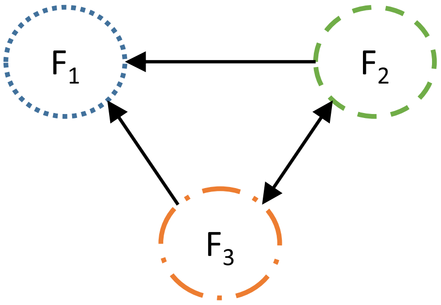


Fig. 2. The vehicle feature graph extracted from the component architecture of Fig. 1.

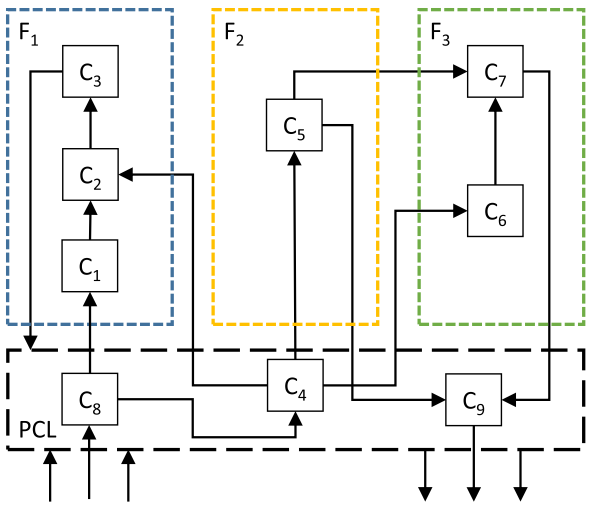


Fig. 3. A layered component architecture with a dedicated platform component layer (PCL) for communal components.

sets of components associated with each feature (lines 3 and 4). If there is a signal flow from a component of the first feature to a component of the second feature, the second feature is added to the set of dependent features of the first feature (line 6). Line 5 refers to a function *signalFlow*(c_1, c_2) that determines if there is a signal flow between c_1 and c_2. This may be implemented differently depending on how the component architecture is represented. In the context of the studies presented in this paper, each component is associated with a set of ports. A signal flow exists between an output and an input port with the same name. Other architecture representations may contain more explicit definitions of signal flow.

2.4. Layered component architectures and communal components

Research and industry have noticed that features and component architectures for embedded systems may become complex and hard to maintain (Dajsuren et al., 2013). Therefore, layered architectures are adopted more frequently for embedded systems to structure software components into layers. The most prominent example of a layered architecture for automotive software is probably AUTOSAR (Fürst and Bechter, 2016), which decouples application software components from base software components within an ECU. However, also within the application software, there is a trend towards structuring the application software components with respect to different dedications (e.g., sensor fusion, controllers, service functionality) (Kugele et al., 2018; Lotz et al., 2019). Fig. 3 shows an example of an application software component architecture that has a dedicated platform component layer (PCL).

Definition 2 (Platform Component Layer (PCL))**.** A *platform component layer* is a dedicated layer within the application software that contains components that are designed to be used in several features.

The question of which components should be part of the PCL and which should be associated with one specific feature is one of the general challenges in designing good software architectures. To support this challenge, we are interested in identifying components that are candidates for the PCL. For this purpose, we characterize specific components as *communal* components.

Definition 3 (Communal Component)**.** A *communal component* is a component that exchanges signals (sending or receiving) with components of features different from the feature of the communal component.

Thus, a communal component is the origin of a feature dependency (see Definition 1). From an architectural point of view, it makes sense to cut out communal components from the context of one specific feature at some point. To acknowledge and leverage the communal character of these components, it is better to treat them as dedicated platform services that are associated with the platform component layer and not with a specific feature. In a former publication (Vogelsang et al., 2016), we reported on evidence collected from developer interviews that communal components cause extra costs for several development activities if they are "hidden" in a specific feature in contrast to an explicit consideration as part of a dedicated platform component layer. Therefore, we distinguish between *implicit* and *explicit* communal components depending on whether the component is associated with a feature or with the platform component layer.

Definition 4 (Implicit Communal Component)**.** An implicit communal component is a communal component associated with a feature and not with the PCL. This means that the component contributes to a feature dependency by exchanging signals with a component that is associated with another feature.

Note that this definition includes both the source component of a feature dependency and the target component. In Fig. 3, C_5 and C_7 are implicit communal components.

In contrast, we call a communal component *explicit* if it is associated with the PCL.

Definition 5 (Explicit Communal Component)**.** An explicit communal component is a communal component that is not associated with a specific feature but with a dedicated platform component layer.

The purpose of the platform component layer is to bundle components that implement functionality important for a number of features. This may include components that provide some general

Table 1
Overview of the study objects.

Characteristics	System 1	System 2	System 3
Type of vehicle	Compact truck	SUV	n/a
Vehicle features			
Original	57	133	217 (155+62)
Cleaned	57	94	144 (116 + 28)
Components			
Original	270	349	959 (882 + 77)
Cleaned	269	325	909 (832 + 77)

signals (e.g., vehicle speed) but also components that collect and process signals for one specific actuator (e.g., different brake demands). In Fig. 3, C_4, C_8 and C_9 are explicit communal components.

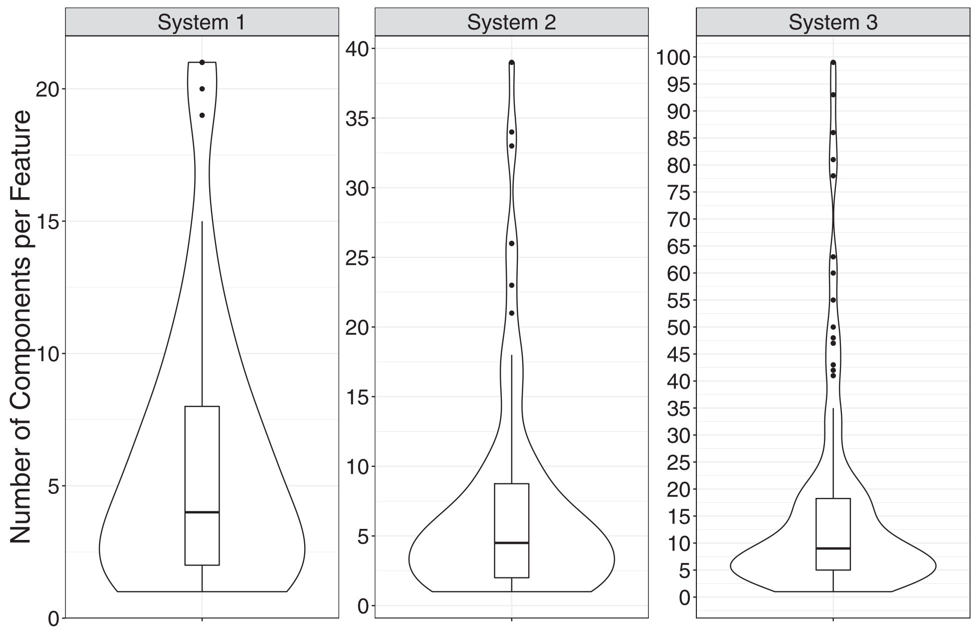
2.5. Study objects

In this article, we answer our research question based on an analysis of three real-world automotive systems from industry. Table 1 summarizes the characteristics of the three examined systems. Before applying the analyses presented in this article, we cleaned the datasets by removing components without any inputs or outputs first and afterward removing all features without any associated components. The numbers presented in Table 1 reflect both the state before (original) and after this preprocessing step (cleaned). In the cleaning step, we removed around 30% of the features from System 2 and System 3. We were told that incomplete data records, such as components without interfaces or features without components, are either legacy/dummy records or the object is not yet fully specified. Therefore, we think it is reasonable to exclude them from our analysis.

The first system we analyze is a vehicle system from MAN Truck & Bus AG, which describes the entire software architecture of a compact truck. The system contains 57 fully specified vehicle features that are realized by an overall of 269 components (after cleaning). The second system is a vehicle system from the BMW Group. Within the component architecture, we focused on the driving dynamics and driver assistance domain. The system comprised 94 vehicle features and 325 components (after cleaning). The third system is also a vehicle system from the BMW Group. The focus of this system is similar to the one of the second system (driving dynamics and driver assistance domain) but the third system is more recent and therefore also more complex. The system contains 144 vehicle features and 909 components (after cleaning).

None of the analyzed systems contained any information about architectural layers. In System 3, however, the system architects distinguish between *regular* features and so-called *service* features. With the concept of service features, the architects acknowledge the fact that some features only exist to provide functionality that is used in other features. This resembles our idea of a platform component layer (see Definition 2) in the sense that all components associated with a service feature can be considered as explicit communal components. We were able to identify the service features in System 3 because the names of the service features start with the prefix “SER”. The system consists of 28 service features and 116 regular features.

Fig. 4 shows the distribution of components per feature. We see that in all systems, the majority of features contains less than 10



Components per feature	System 1 (57 features)	System 2 (94 features)	System 3 (116 features)
Maximum	21	39	99
Median	4	4.5	9
Std. Dev.	5.3	7.6	20.0
Minimum	1	1	1

Fig. 4. Distribution of components per feature.

Table 2
Coupling and cohesion of features in the study objects and in reference systems based on metrics defined by Dajsuren et al. (2013); Dajsuren (2015).

Metric	System 1	System 2	System 3	Dajsuren et al. (2013)
Subsystem cohesion metric				
Max	1.00	1.00	1.00	0.20
Mean	0.21	0.27	0.16	0.08
Min	0.00	0.00	0.01	0.03
Coupling between subsystems				
Max	30	91	157	3
Mean	4.7	30.8	64.3	1.6
Min	0	0	0	0

Table 3
Positions of components in features from Fig. 1.

F.	C.	Input distance	Output distance	Sum	Position
F_1	C_1	$\rightarrow C_1$: 1	$C_1 \rightarrow C_4 \rightarrow$: 2	3	1/3=0.33
	C_4	$\rightarrow C_1 \rightarrow C_4$: 2	$C_4 \rightarrow$: 1	3	2/3=0.66
F_2	C_2	$\rightarrow C_6 \rightarrow C_3 \rightarrow C_2$: 3	$C_2 \rightarrow$: 1	4	3/4=0.75
	C_3	$\rightarrow C_6 \rightarrow C_3$: 2	$C_3 \rightarrow C_2 \rightarrow$: 2	4	2/4=0.50
	C_6	$\rightarrow C_6$: 1	$C_6 \rightarrow C_3 \rightarrow C_2 \rightarrow$: 3	4	1/4=0.25
F_3	C_2	$\rightarrow C_5 \rightarrow C_2$: 2	$C_2 \rightarrow$: 1	3	2/3=0.66
	C_5	$\rightarrow C_5$: 1	$C_5 \rightarrow C_6 \rightarrow$: 2	3	1/3=0.33
	C_6	$\rightarrow C_5 \rightarrow C_6$: 2	$C_6 \rightarrow$: 1	3	2/3=0.66

components (for System 1 and System 2 even less than 5). The larger the systems, the larger single features are. The largest feature in System 3 consists of 99 components.

Inspired by the work of (Dajsuren et al., 2013; Dajsuren, 2015), we report basic metrics on coupling and cohesion of features in the three examined systems in Table 2. Dajsuren et al. proposed these metrics (besides others) for assessing the modularity of an automotive architecture (Dajsuren, 2015). More specifically, the Subsystem Cohesion Metric (SCM) measures the inter-relation of the blocks within a subsystem (see Dajsuren, 2015 for the exact definition). Coupling Between Subsystems (CBS), on the other hand, measures the number of subsystems that is influenced by a subsystem or that influences the subsystem. Dajsuren et al. applied these metrics in the context of Simulink models (Dajsuren et al., 2013) and provided results for ten models that they analyzed. The measurements of these ten reference systems are provided in the last column of Table 2. From the table, we see that the feature cohesion in our examined systems is similar, yet much higher than in the Simulink models examined by Dajsuren et al. There are large differences in the coupling of features as the systems that we examine have many more dependencies than the subsystems considered in the Simulink models.

The companies that provided the systems manage the features and components of their systems in a company-specific tool. We were able to export the relevant data from this tool to use it for further processing and applying our dependency extraction algorithm. In all three cases, the exports contained the set of features, the set of components associated with each feature, and a description of the interface of each component in terms of input and output ports.

We performed our analysis in close collaboration with the companies and discussed our results with them.

3. Extent of feature dependencies in automotive systems

In this section, we answer the question of how many feature dependencies exist in the analyzed systems and how the dependencies are distributed among the features. We structure this section by two research questions.

3.1. Research questions

RQ1: To what extent do dependencies between vehicle features exist?
We focus on dependencies in the sense that the behavior of a vehicle feature is not solely dependent on its primary inputs but also on the input of another vehicle feature (see Section 2).

RQ2: How are dependencies distributed over the vehicle features?
We are interested in whether dependencies are equally distributed over all vehicle features or if there are vehicle features that are more central with respect to dependencies. In addition, we want to know whether the *position* of a component in a feature has an impact on the number and type of associated feature dependencies. By position, we refer to the position in the chain of processing within a feature (e.g., components related to sensing, processing, or actuation).

3.2. Data collection

To answer the research questions, we extracted the vehicle feature graphs for the three analyzed systems as defined in Section 2 by means of a simple tool, written in Java[1]. The tool parses the exported dataset from the company's data backbone containing a list of vehicle features associated with a set of components. The tool extracts the feature dependencies according to the definition given in Section 2 and outputs a `.csv` file with the found dependencies. The extraction is fully automated and the complexity of the algorithm is quadratic in the number of vehicle features and components. For the observed systems, the extraction took less than 3 seconds on a standard laptop.

3.3. Analysis procedures

To answer the research questions, we collected the following measures:

For RQ1, we analyzed the vehicle feature graph to assess the ratio of vehicle features that depend on another vehicle feature and to count the number of incoming and outgoing dependencies between vehicle features. This gives an impression of the extent of feature dependencies in realistic systems.

For RQ2, we measured the dependency fan-in and fan-out for all vehicle features of the vehicle feature graph to see whether dependencies are distributed equally or if certain vehicle features are more central than others. Thus, we obtain information about the distribution of feature dependencies in real automotive software systems. In addition, we relate the number of feature dependencies associated with a single component to the position of this component in the processing chain of a feature. For this purpose, we compute the longest distance from a component to any input and output of the associated feature. We add the distances for each component and compute a normalized position indicator by dividing the input distance to the maximum sum of input and output distance of any component in the considered feature. A component without any input or output channel within the considered feature is considered as source or sink respectively and the distance is set to 0. Table 3 shows the results of this position determination for the architecture given in Fig. 1. The resulting positions indicate a kind of topological order of the components within the processing chain of a feature. Small values point to components at the beginning of a feature processing chain (e.g., sensor processing) while values close to 1 point to output-oriented components (e.g., actuation).

[1] The tool is available: https://github.com/andivogelsang/FeatureDependencyAnalyzer.

Table 4
Extent of dependencies in the vehicle feature graph.

	System 1 (57 features)		System 2 (94 features)		System 3 (116 features)	
Features...	#	%	#	%	#	%
with incoming dependencies	37	64.9	81	86.2	110	94.8
with outgoing dependencies	30	52.6	72	76.6	101	87.1
with incoming and outgoing dependencies	27	47.4	68	72.3	98	84.5
without dependencies	17	29.8	9	9.6	3	2.6

3.4. Validity procedures

To ensure internal validity for System 1, we analyzed the system under investigation at a stage where it had already been subject to an architectural review. Thus, design flaws and misconceptions within the analyzed model should be reduced. Additionally, we presented and discussed the results with the developers, who assured that our results are valid and reasonable.

We analyzed System 2 at a final stage of the development process where it was already subject to several architectural reviews and testing procedures. Therefore, errors and misconceptions in the component architecture can nearly be ruled out. To further ensure validity, we presented and discussed the results with experts from the company, who assessed the dependencies we found with respect to their plausibility.

We did not have the chance to present and discuss our results for System 3 with the actual developers. However, we analyzed the system at a stage where it had already been subject to several architectural reviews. In addition, we only considered the features of System 3 that were not characterized as service features.

3.5. Study results

In this section, we present the study results. They are structured according to the defined research questions.

3.5.1. Extent of dependencies (RQ1)

Table 4 summarizes the results of the analysis concerning the extent of dependencies in the three analyzed systems.

Analyzing the vehicle feature graph of System 1, we found 136 dependencies between the 57 vehicle features. 17 out of the 57 vehicle features were completely independent of any other vehicle feature and did not have any influence on other vehicle features. 37 vehicle features depend on other vehicle features (i.e., they have incoming dependencies) and 30 vehicle features influence other vehicle feature (i.e., they have outgoing dependencies).

Analyzing the vehicle feature graph of System 2, we found 1451 dependencies between the 94 vehicle features. Only 9 out of the 94 vehicle features were completely independent of any other vehicle feature. 81 vehicle features depend on other vehicle features and 72 vehicle features influenced other vehicle feature.

Analyzing the vehicle feature graph of System 3, we found 3728 dependencies between the 116 regular vehicle features. Only 3 out of the 116 vehicle features were completely independent of any other vehicle feature. 110 vehicle features depend on other vehicle features and 101 vehicle features influenced other vehicle features.

We did not find any specific patterns in the features without any dependencies. Some features were related to display functionality, others were related to lighting functions. The features with most feature dependencies included complex driver assistance features such as Adaptive Cruise Control (ACC) or traffic jam assistant (ACC + lane keeping) and engine-related features such as the start-stop feature.

Table 5
Plausibility and awareness of all analyzed feature dependencies in System 2.

Dependencies	known	unknown	All
Incoming			$n = 63$
All	38%	62%	
plausible	37%	59%	95%
implausible	2%	3%	5%
Outgoing			$n = 37$
All	49%	51%	
plausible	49%	30%	78%
implausible	0%	22%	22%
Incoming + Outgoing			$n = 100$
All	42%	58%	
plausible	41%	48%	89%
implausible	1%	10%	11%

3.5.2. Distribution of dependencies (RQ2)

The extent of the dependencies shows that dependencies between vehicle features are distributed all over the system. However, some vehicle features are more central in the sense that they have a large number of dependencies to other vehicle features. Table 5 illustrates this result in violin plots and minimum, maximum, and median in a table. The figure shows that, for System 1, vehicle features have a maximum of 23 other vehicle features that they influence, whereas one vehicle feature depends on up to 10 other vehicle features. On average (median), each vehicle feature depends on one other vehicle feature and influences one other vehicle feature.

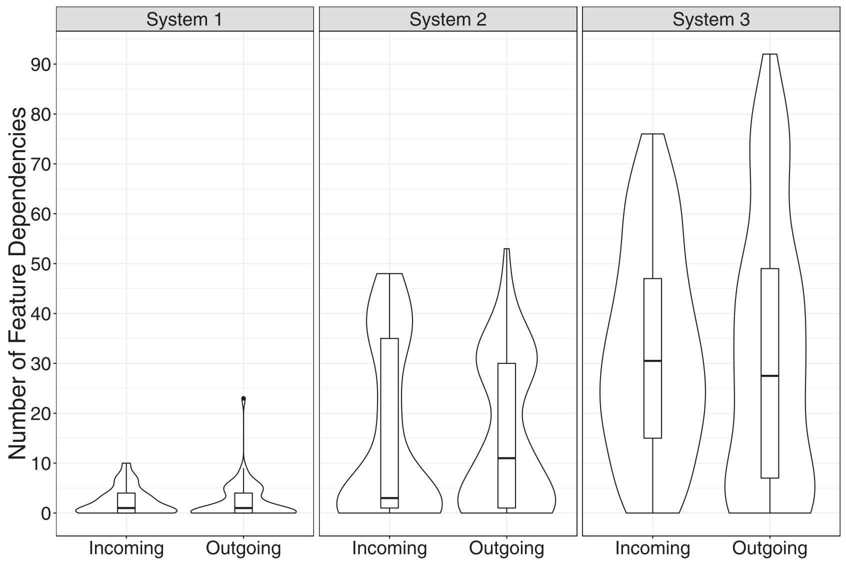
For System 2, these numbers are higher. A vehicle feature in this system depends on up to 48 other vehicle features, whereas one vehicle feature has a maximum of 53 other vehicle features that it influences, which accounts for 56% of the vehicle features. Most of the features have at least 3 features they depend on and at least 11 features they influence.

We found the largest variance of feature dependencies in System 3. A vehicle feature in this system depends on up to 76 other vehicle features, whereas one vehicle feature influences up to 92 other vehicle features, which accounts for 79% of all vehicle features. On average (median), each vehicle feature depends on 30 other vehicle features and influences 27 other vehicle features.

The distributions of dependencies in the three systems do not indicate to follow a certain statistical distribution. While for System 1, the distribution of feature dependencies fits a classical *long-tail* distribution (i.e., it is right-skewed), the distribution of feature dependencies for System 2 looks more like a bimodal distribution, and the distribution in System 3 is almost uniform (with a slight right-skew). We take this as an indication that there is not an implicit underlying working principle that constitutes a reason for feature dependencies.

The intermeshed structure of the features becomes particularly visible when illustrating the dependencies in a *Bundled Edge View* (Holten, 2006) (see Fig. 6).

Fig. 7 shows the positions of components in a feature and the associated number of incoming and outgoing dependencies. As ex-



Feature Dependencies	System 1 (57 features)		System 2 (94 features)		System 3 (116 features)	
	In	Out	In	Out	In	Out
Maximum	10	23	48	53	76	92
Median	1	1	3	11	30.5	27.5
Std. Dev.	2.7	3.8	17.2	14.9	20.7	26.6
Minimum	0	0	0	0	0	0

Fig. 5. Distribution of feature dependencies per feature.

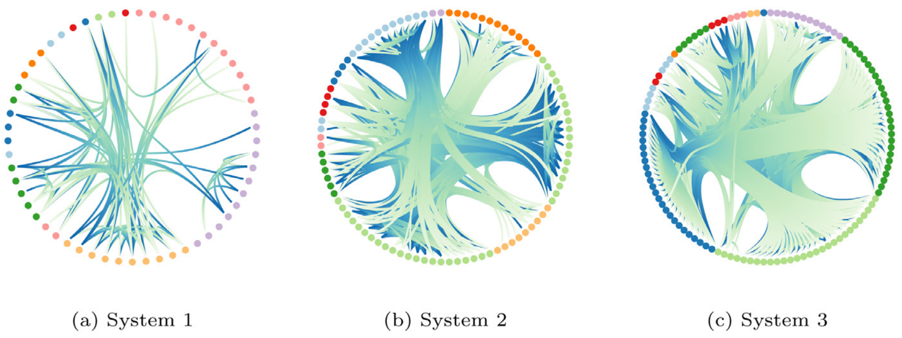


Fig. 6. Features dependencies visualized as Bundled Edge View. The outer ring represents the hierarchy of features. Each dot on the inside of the outer ring represents a feature. The edges indicate a dependency between two features.

plained before, a position indicator close to 0 indicates components at the beginning of a feature processing chain, whereas values close to 1 indicate a position at the end of the processing chain. The figure shows that the distribution is different in the three systems, however, we can still observe some common trends.

Components with incoming dependencies are mostly located in the middle of a feature processing chain. This may indicate that information from other features is needed in the central "business logic" of a feature (e.g., to validate results or to check side-conditions of other features). Components with outgoing dependencies are located more towards the beginning of a feature processing chain. This makes sense to some degree as components at the beginning of a feature are often related to sensor handling and data preprocessing. Both tasks may also be of interest in other features.

In System 3, the extent of outgoing dependencies in components at the beginning of a feature is not as large as in the other systems. A reason for this phenomenon could be the introduction of service features in System 3 (see Section 2.5). Since the service features are excluded from the analysis of System 3, it is possible that mostly components from the beginning of a feature processing chain have been transferred to service features. As a result, the number of dependencies related to early components is reduced in System 3. We investigate this effect in more detail in RQ4.

3.6. Conclusion

Our analysis shows that the majority of features in the examined systems cannot be considered in isolation. The larger the systems are, the more intermeshed and dependent the features become. Especially for System 3, it is hard to consider the feature structure as a proper functional breakdown because most features rely on more than one-quarter of all features and influence similarly many other features. In addition, our analysis shows that

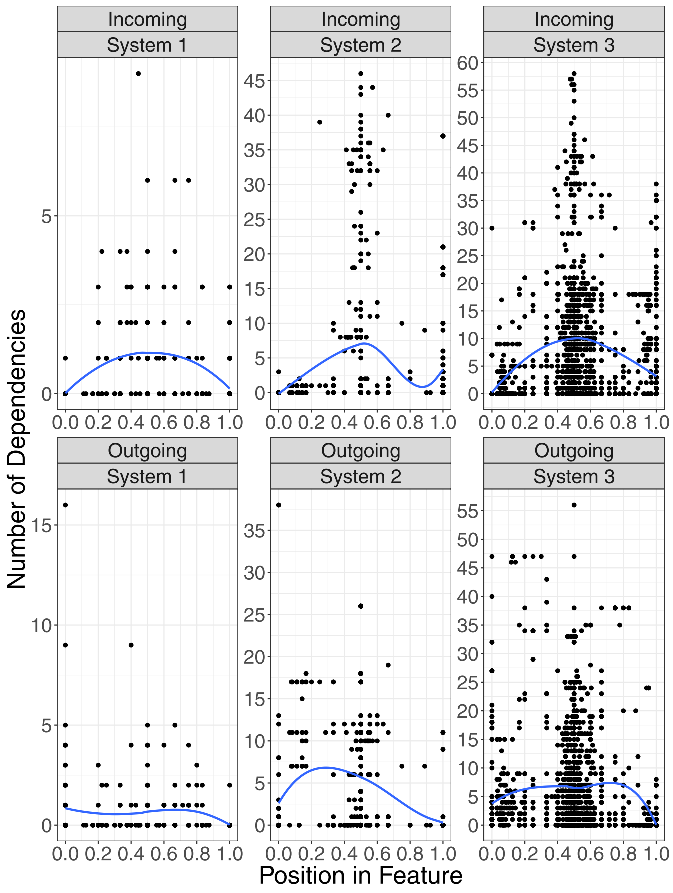


Fig. 7. Number of dependencies per component in relation to its position in the processing chain of a feature.

components at the beginning a feature processing chain are more subject to outgoing dependencies, while components in the middle and towards the end of a feature are stronger associated with incoming dependencies. An interesting case is System 3, in which the architects have started to define service features. This may have already impacted the observed distribution of dependencies. We investigate this in more detail in RQ4. From our results, we conclude that feature dependencies are a phenomenon that developers should be aware of when they develop features and design the overall architecture of automotive software systems. Whether this is the case is subject to the next section.

4. Awareness of feature dependencies in automotive systems

As explained in the introduction of this article, automotive organizations are largely structured with respect to features and their domains. Given the large number of feature dependencies that we found in the examined systems, we are interested in the question whether the developers of the features are aware of these dependencies and how dependencies influence the development of the features and the component architecture. We address this objective by answering two questions that emerged from the specifics of the cases that we considered.

4.1. Research questions

RQ3: To what extent are developers aware of feature dependencies?
Developers of automotive systems are not necessarily aware of existing dependencies. We want to identify existing feature dependencies that are unknown to developers. For System 2, we had the chance to examine and discuss a subset of the extracted feature dependencies with the developers in detail. Therefore, we addressed this research question in the context of System 2.

RQ4: Are feature dependencies more frequent in service features compared with other features?
As described in Section 2.5, System 3 contains regular features and so-called service features. We expect that feature dependencies appear more frequently in service features compared with the regular features because service features are designed to support other features. We were told that the definition of service features is done based on expert opinions and experience.

Both research questions help us to understand the awareness of feature dependencies in current working practices.

4.2. Data collection

To answer RQ3, we performed interviews with four engineers from the company of System 2, who are involved in the development of features and component architectures. We confronted the experts with a sample of feature dependencies that we found in our analysis and that matched their area of responsibility. In order to get representative results from the interview partners, we selected one expert from each area within the domain of driving dynamics and driver assistance. These areas are lateral, longitudinal, and vertical dynamics as well as driver assistance features. The experts each were responsible for a number of 12–46 vehicle features.

We let the experts classify each dependency into the following categories:

- **plausible/implausible:** A dependency is considered as *plausible* if the expert finds a functional or physical explanation for this dependency. If the expert has no functional or physical explanation for this dependency, it is considered as *implausible*.
- **known/unknown:** A dependency is considered as *known* if the expert was aware of this dependency prior to the interview. If the expert was not aware of this dependency prior to the interview, it is considered as *unknown*.

In total, we discussed 100 feature dependencies in depth (i.e., 6.7% of all dependencies).

To answer RQ4, we included the service features into our analysis of feature dependencies and treated them as if they were regular features. We performed the same feature dependency analysis as presented in Section 3 on the set of all features (regular and service features) and compared the two types of features afterwards.

4.3. Analysis procedures

For RQ3, we counted the number and ratio of feature dependencies in System 2 for each combination of category values, leading to a 2×2 matrix with the two categories as dimensions. We interpret the ratio of *plausible* feature dependencies as an indicator for the validity of our feature dependency extraction approach and the ratio of *known* feature dependencies as an indicator for the awareness of feature dependencies in general.

For RQ4, we performed hypothesis tests with H_0: *Service features and regular features have the same number of feature dependencies* and the alternative hypothesis H_1: *Service features have more feature dependencies than regular features*. We performed this test for the total number of dependencies (incoming and outgoing) as well as individually for the number of incoming and for the number of outgoing feature dependencies. We used a Shapiro–Wilk's test to check whether the number of feature dependencies is normally distributed in the sets of regular and service features. It turned out that they are not normally distributed (p-values

< 0.05). Therefore, we used an unpaired two-sample Wilcoxon test (a.k.a. Mann-Whitney U test) for the hypothesis test with a confidence level of 0.95.

4.4. Study results

4.4.1. Awareness of dependencies (RQ3)

Table 5 summarizes the results of the expert interviews that we conducted for System 2 in order to assess the plausibility and awareness of the analyzed feature dependencies. The table shows the results for incoming, outgoing, and all dependencies separately.

The results indicate that our analysis produced reasonable results as only 11% of the examined feature dependencies were considered as *implausible*, i.e., the dependencies were a result of our analysis but the experts considered them as not correct or at least they were not able to give account for them. Such cases included dependencies between features that were very similar to each other or they were in fact alternatives. The interviewees stated that there should not be a dependency between those features as they would never appear in one specific product. Yet, they were not able to explain why these dependencies still exist in the data. Three further feature dependencies that were considered implausible relate to one specific damping feature.

Of the 100 feature dependencies that we examined in the interviews, 42% were known to the experts and 58% were unknown (see Table 5). The largest group of feature dependencies that we examined was the group of unknown but plausible feature dependencies, i.e., the experts were not aware of the dependency between the features but when examining the affected signals and components they found reasonable explanations for them. We did not find specific patterns of particular types of dependencies that are more prone to be unknown to the developers. One examined dependency was considered as known but implausible as the expert was aware of it but had no explanation why this dependency exists.

The detailed results show that there is a difference in the awareness of incoming and outgoing feature dependencies. Our interview participants considered incoming dependencies plausible by a higher ratio than outgoing dependencies (95% vs. 78%). One explanation may be that it is easier for developers to assess the plausibility of dependencies if they relate to signals that are used in the feature of the developer (i.e., incoming dependencies). In contrast, it may be harder to assess whether a signal that is produced by the feature of the developer is used in another feature (i.e., outgoing dependency). On the other hand, our participants considered relatively more incoming feature dependencies as unknown than outgoing feature dependencies (62% vs. 51%).

4.4.2. Dependencies of service features (RQ4)

Fig. 8 shows the distribution of feature dependencies in the sets of regular and service features in System 3. As expected, the service features have a higher number of feature dependencies compared with the regular features. By adding the 28 service features to the analysis of the 116 regular features, the total number of feature dependencies almost doubles (7,071 vs. 3728 feature dependencies). We were able to reject H_0 in favor of H_1 in the case of all dependencies (p-value: 0.0003) and outgoing dependencies (p-value: 0.014). We were not able to reject H_0 in favor of H_1 for the case of incoming dependencies (p-value: 0.087). The median number of feature dependencies in service features is 63% greater than in regular features.

The distribution of dependencies in the sets of regular and service features indicates that the service features are in line with their purpose. The distribution is left-skewed, i.e., the mass of the distribution is concentrated on higher numbers of dependencies. In comparison, the distribution of dependencies in regular features is right-skewed, i.e., the mass of the distribution is concentrated on lower numbers of dependencies.

Although we did not have information about the details of the service features, the names already indicate the purpose of single service features. 15 out of the 28 service features have the term “provisioning” in combination with a specific signal in their name (e.g., *SER_provisioning_steeringangle*). This indicates that these features encapsulate behavior related to the provisioning of a specific information. Another 10 service feature names contain the term “actuation” in combination with a specific signal (e.g., *SER_actuation_blinking*). Such service features may collect control commands from different features, bundle (and maybe prioritize) them, and then forward the signal to a controller. The difference between these two groups of service feature is also reflected by the number of incoming and outgoing dependencies. While the “provisioning” service features have more outgoing than incoming dependencies in general, for “actuation” service features, it is the other way around.

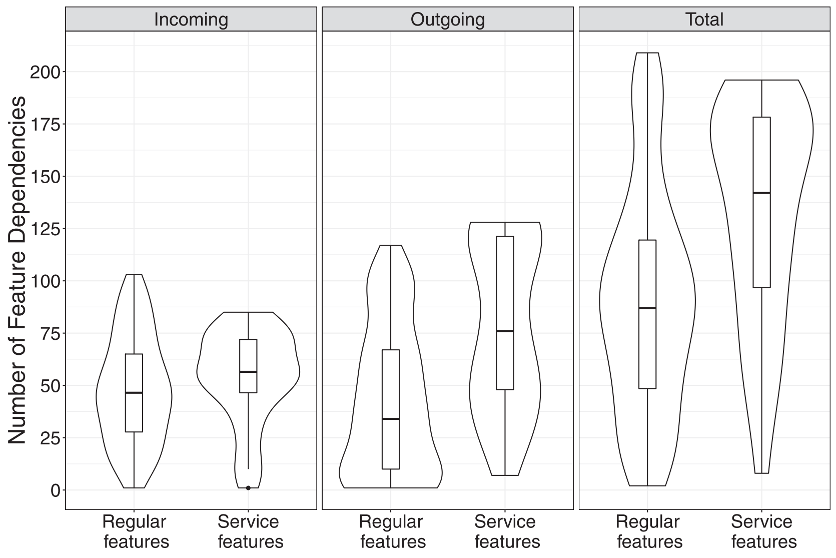
4.5. Conclusions

Our analysis shows two things: (1) Developers of features are not aware of a large part of dependencies between features. In order to implement their features, the developers use signals available on the bus systems without considering where these signals come from. Similarly, they are not aware of which other features are using the signals that are produced in the feature of a developer. In the current way of how features are developed in the examined companies, unawareness about outgoing dependencies poses a challenge because signals are changed without a proper change-impact analysis. As a result, errors that arise from a changed signal are detected late when features are integrated. With the help of an analysis as presented in this paper, developers are aware of outgoing dependencies much earlier and a change-impact analysis is possible.

(2) On the other hand, the comparison of regular and service features indicates that feature dependencies might play an important role in structuring the system. Our results show that features with a large number of dependencies are candidates for serving as service features that have a different role than regular features in the vehicle architecture. However, our results also indicate that currently, outgoing feature dependencies play a larger role for service features than incoming feature dependencies. This is also in line with the results of RQ2, where incoming dependencies in System 3 appeared in components more towards the middle and end of a feature processing chain. Components at the beginning of the processing chain are strongly associated with outgoing dependencies. A reason for this may be that it is easier to identify commonly used signals and then define the producing feature as a service feature compared to identifying features that collect many signals to merge them. In the next section, we present an approach that systematically considers incoming and outgoing dependencies to suggest candidates for service features.

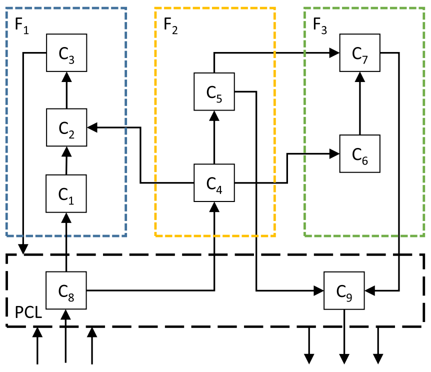
5. Dependency-based refactoring

Based on the findings of the last sections, we conclude that feature dependencies are a phenomenon that is omnipresent in automotive architectures but that developers are not very aware of and that does not play a major role when considering architectural decisions such as deciding which functionality to provide as platform services. In addition, we recognized in discussions with developers that feature dependencies do not necessarily exist right away from the beginning. In most cases, feature dependencies emerge over time, when signals of components that were initially specified only for one specific feature are (re)used by components of

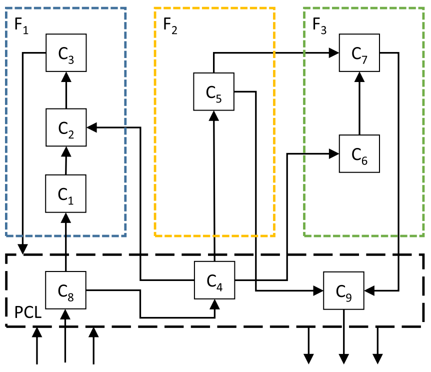


Feature Dep.	Incoming		Outgoing		Total	
	Reg.	Ser.	Reg.	Ser.	Reg.	Ser.
Maximum	103	85	117	128	209	196
Median	46.5	56.5	34	76	87	142
Std. Dev.	25.9	22.5	35.4	40.2	54.0	52.5
Minimum	1	1	1	7	2	8

Fig. 8. Distribution of feature dependencies for regular and service features of System 3.



(a) Before refactoring.



(b) After refactoring

Fig. 9. Communal component C_4 is refactored by shifting it into the dedicated platform component layer (PCL).

other features. Component signals are transmitted via bus systems, which makes the information available to all components connected to that bus. Therefore, it is a common practice of developers to (re)use any signal that is available on the bus system to implement or adapt a feature, regardless of the origin of that signal. By this practice, a component that was originally designed for one specific feature becomes relevant for other features as well. Thus, a component may over time become an implicit communal component (see Section 2.4).

5.1. Approach: Dependency-based refactoring

In the context of this study, we make an implicit communal component explicit by extracting it from its original feature and shifting it into the PCL. Fig. 9b shows a refactored version of the architecture of Fig. 9b, where the implicit communal component C_4 is made explicit by shifting it to the PCL. We do not consider signal flow from components of the PCL to components of a feature as a feature dependency. Therefore, making implicit communal components explicit reduces the number of feature dependencies in the system. For example, by the refactoring shown in Fig. 9b, we removed the feature dependency between F_1 and F_2. Of course, component C_2 still reads the signal provided by C_4, but now C_4 is not associated with a feature but with the platform component layer. That means, the dependency between C_2 and C_4 does not completely disappear but it changes from a feature dependency to a dependency between a feature and the PCL. This change has a positive impact on efforts for several development activities. In fact, implicit communal components can be characterized as technical debt (Vogelsang et al., 2016).

We propose this dependency-based refactoring approach as a systematic and objective way to decide which communal components are candidates for being refactored and to reduce the technical debt in the architecture. To evaluate how well this approach reflects reality, we applied the approach to the three case study systems.

5.2. Research questions

Our research goal is to assess the extent of *implicit communal components* within real-world automotive systems and examine the effect of a dependency-based refactoring approach on the number of feature dependencies. For this purpose, we follow two research questions.

RQ5: How many implicit communal components exist in real-world automotive systems?
We want to assess the extent of implicit communal components in real-world automotive systems. The answer to this question indicates the relevance of considering implicit communal components as technical debt.

RQ6: What is the effect of dependency-based refactoring on the number of feature dependencies?
We want to understand if and how our dependency-based refactoring approach leads to a reduction in feature dependencies. The answer to this question may indicate whether it is reasonable to consider dependency-based refactoring for architectural evolutions of automotive systems.

5.3. Data collection and analysis

To answer RQ5, we calculated the number of implicit communal components and resulting feature dependencies within the component architecture based on Definitions 1 and 4. In System 3, we excluded all service features for this analysis to consider the fact that these features are already intended to serve as platform service features.

To answer RQ6, we implemented and analyzed a greedy algorithm that removes, in each iteration, the single implicit communal component that contributes to the largest number of feature dependencies. This refactoring operation corresponds to the idea of shifting an implicit communal component to a dedicated platform component layer, i.e., the component does no longer contribute to any feature dependency. The algorithm terminates when all implicit communal components have been refactored and thus no feature dependency exists anymore. As a result of one refactoring step, a number of feature dependencies are removed and consequently also a number of implicit communal components may become “normal” components (when the removed feature dependencies are the only dependencies a communal component contributes to). Thus, the algorithm is steadily decreasing the number of feature dependencies with each refactoring of an implicit communal component.

5.4. Study results

5.4.1. RQ5: Number of implicit communal components

Table 6 depicts the results of RQ5. For the small truck (System 1), our automated analysis returned an overall of 97 implicit communal components, which accounts for 36% of all components. These 97 communal components contribute to an overall of 136 feature dependencies between the 57 features of the system. For the SUV (System 2), we identified 175 implicit communal components, which accounts for 46% of all components. The communal components contribute to an overall of 1451 feature dependencies between the 94 features of the system. For System 3, we identified 504 implicit communal components, which accounts for 60% of all components (not counting components already assciated with service features). The communal components contribute to an overall of 3728 feature dependencies between the 116 features of the system that are not service features.

Interpretation: The goal of RQ5 was to assess the extent of implicit communal components in real-world automotive systems to indicate the relevance of considering implicit communal components as technical debt. The presented results show that in the examined systems 36–60% of all components are implicit communal components. Thus, refactoring all implicit communal components at once is not realistic and, therefore, it is useful to consider implicit communal components as technical debt that should be removed when the refactoring pays off.

Table 6
Overview of the study results for RQ5 .

Results for RQ5	System 1	System 2	System 3
Implicit communal components	97 (36%)	175 (46%)	504 (60%)
Feature dependencies	136	1,451	3,728

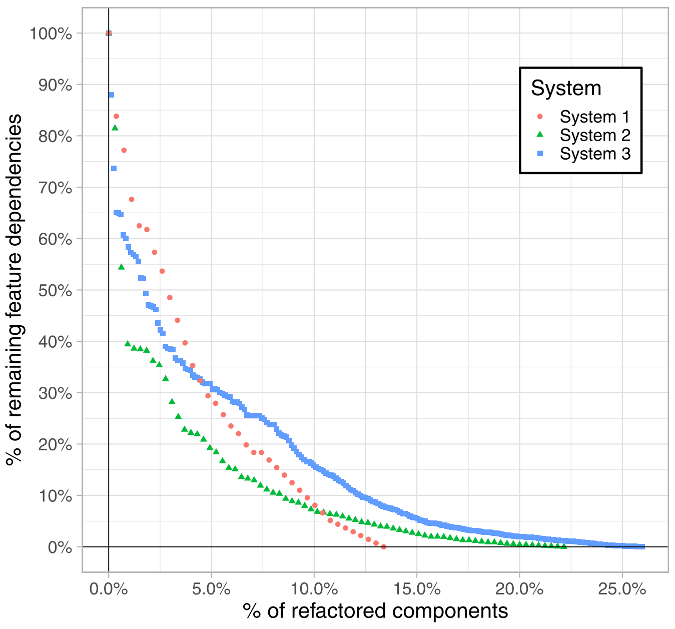


Fig. 10. Remaining feature dependencies after successively refactoring implicit communal components.

5.4.2. RQ6: Effect of dependency-based refactoring on feature dependencies

Fig. 10 shows the remaining percentage of feature dependencies within the three systems after each iteration of the mentioned greedy algorithm that successively refactors the component that contributes to the largest number of feature dependencies. The figure shows that, for all systems, the number of feature dependencies decreases strongly after refactoring only a few components. In fact, to remove 90% of the feature dependencies, we need to refactor less than 12.5% of the components. To remove all feature dependencies, we need to refactor 13% of components for System 1, 22% for System 2, and 26% for System 3.

In addition to the number of remaining feature dependencies, which resembles a reduction of coupling of features, we also analyzed the corresponding cohesion. A reduction of coupling may come with the risk of lower cohesion because features may get pulled apart. However, for the three examined systems, we rather observe a slight increase in cohesion. In Section 2.5, we reported the original mean cohesion values of the systems in terms of the Subsystem Cohesion Metric (Dajsuren, 2015). This measure of cohesion increased during the refactoring simulation from 0.21 to 0.24 for System 1, from 0.27 to 0.31 for System 2, and from 0.16 to 0.22 for System 3. That means that the refactoring operations in the examined systems do not have a negative influence on the cohesion of features.

Interpretation: The goal of RQ6 was to understand how single communal components contribute to the number of feature dependencies to justify an individual assessment of components with respect to their impact on feature dependencies. The presented results show that the contribution to feature dependencies strongly differs between single communal components. Therefore, refactoring some communal components has a much higher impact on feature dependencies than refactoring others. Assuming that the refactoring operation is equally costly for all implicit communal components, this means that there are some communal components for which the refactoring saves much more effort compared to others. This results in a trade-off between costs for the refactoring step and benefits gained from removing the corresponding feature dependencies. We characterized this trade-off as a type of technical debt in earlier publications and identified a number of cost factors that need to be considered when assessing the trade-off (Vogelsang et al., 2016). By considering this trade-off, it should be clear that it is usually not desirable to refactor *all* implicit communal components. Components should only be refactored if the (short- and long-term) costs for refactoring are lower than the costs for an implicit communal component. This decision may also be influenced by the current state of the PCL. Shifting more and more components to the PCL may make the PCL large and complicated, which increases the costs for maintaining the PCL. System architects must find the sweet spot between a manageable PCL and a low number of feature dependencies. Our analysis does not provide a definitive answer to this sweet spot but we would consider a refactoring of 10% of components as reasonable as this would result in a decrease in feature dependencies of almost 90% while the PCL consists of only 26–83 components for the analyzed systems.

6. Threats to validity

We structure and discuss the threats to validity along the three aspects that we have addressed in this article.

6.1. Extent of feature dependencies

Construct validity: Our analysis relies on our definition of feature dependencies as given in Section 2. There may be other definitions of feature dependencies that may also be used to answer the RQs and that may lead to different results. Besides the explicitly modeled dependencies that are in the focus of this study, there may also be dependencies between vehicle features that occur when functions are implicitly connected by a feedback loop through the environment. Considering also such dependencies may additionally increase the number of detected feature dependencies. The specification and detection of these dependencies would require considering a system together with a precise specification of its context.

Internal validity: A threat to the internal validity is the fact that the analyzed models are already a realization/implementation of the vehicle features. Dependencies might thus be a consequence of design decisions made by developers and not a necessity of the vehicle feature itself. This effect can be seen as an instance of the *optional feature problem* (Kästner et al., 2009), which addresses that the implementation of features may be dependent, although the actual features are independent in the problem domain.

External validity: A threat to the external validity is that we performed this study in a development and tooling context specific to the two companies that provided the examined systems. This context might not be transferable to other companies or domains. However, from our experience, we are confident that the definition of vehicle features, which are implemented by a network of functional blocks or components, is common in the development of automotive software systems.

6.2. Awareness of feature dependencies

Construct validity: We determine awareness from two angles. One is a direct measure, where we ask developers whether they know certain feature dependencies, the other is an indirect measure, where we examine the expected effect of feature dependencies toward the definition of so-called service features. Besides these two measures, there may be other measures to determine awareness (cf. Timmermans and Cleeremans, 2015). In addition, awareness may not be a good indicator to reveal actual development problems caused by feature dependencies. In earlier work (Vogelsang et al., 2016), we made a first attempt to relate the phenomenon of feature dependencies to development activities that become more costly in the presence of feature dependencies. This activity-oriented characterization may lead to a more operational characterization of the challenges related to feature dependencies (cf. Femmer and Vogelsang, 2019).

Internal validity: We use *triangulation* and assess the concept of awareness from two different angles as explained above. By this, we aim to reduce the threats to validity related to qualitative research (reactivity, researcher bias, and respondent bias (Robson, 2002)).

External validity: Similar to our analysis of the extent of feature dependencies, our conclusions for the awareness are based on the observations of only two systems (System 2 and System 3) in the specific context of one company. Thus, it is possible that other developers of other automotive companies are more (or less) aware of feature dependencies due to better tooling, methodologies, or just because the systems they develop are less complex. We think that the level of complexity should be similar for other automotive OEMs, however, it may be the case that feature dependencies are not so much of a problem for automotive supplier companies who usually build smaller subsystems.

6.3. Dependency-based refactoring

Construct validity: The study on the effect of applying the refactoring operation is based on a simulation of refactoring steps. There was no actual refactoring of those systems that could be tested and evaluated whether it preserved the behavior of the previous versions.

Although we consider the technical debt metaphor as a useful instrument to explain the effects of communal components, we are not sure whether an operationalization can provide quantitative measures that allow for an exact prediction of cost savings. It is an open issue whether it is possible to quantify the cost factors provided in this paper and how to weight them.

Internal validity: A limitation of our study is that we only considered one type of refactoring. There may be other possibilities of refactoring implicit communal components with a different cost structure (e.g., making implicit communal components explicit by labeling them inside a feature and not moving them to the PCL).

External validity: From a research methodological point of view, the sample size of our study poses a threat to the validity of the results. We answered the RQs on the basis of examining only three system instances.

7. Related work

Apel et al. (2013b) explored feature interactions in real-world systems and characterized them by two dimensions: order and visibility. The order of a feature interaction is defined as the minimum number of features (minus one) that need to be activated to trigger the interaction. For visibility, the authors distinguish between *external* and *internal* feature interactions. External feature interactions may appear at the level of externally-visible behavior. They are subdivided into *functional* interactions, which address interactions violating the functional specification of a system and *non-functional* interactions, which address interactions influencing non-functional properties (e.g., performance, memory consumptions, or energy consumption). Internal feature interactions, on the other hand, may appear at the level of the internal properties of a system. They are subdivided into *structural* interactions, which can be detected by static analysis of the syntactic program structure and *operational* interactions, which can only be detected by more sophisticated analyses (e.g., control or data flow analysis). In their article, Apel et al. (2013b) give preliminary results considering the detection and classification of feature interactions in four real-world systems: Linux, BusyBox, GCC, and Apache. Feature interactions occurred in all systems, and the authors found interactions of all kinds, including structural, operational, functional, and non-functional interactions. The internal feature interactions outnumbered the external feature interactions found. Ribeiro et al. report similar results when examining preprocessor-based code (Ribeiro et al., 2011). They found that 66% ± 18.5% of the methods with directives have dependencies. In our study, we found feature dependencies in more than 70% of all features and more than 36% of components. We were interested in *external functional* feature interactions (Apel et al., 2013b) and call them feature dependencies. In Section 3, we investigate *structural* (i.e., internal) interactions in an automotive system to derive feature dependencies. Our results support the presumption of Apel et al. (2013b), who assume a relation between internal and external feature interactions: *"We [the authors] believe that there may be systematic correlations between externally-visible and internally-visible interactions, which is a major motivation for our endeavor to explore and understand the nature of feature interactions."*

In a study by Kästner et al. on the optional feature problem (Kästner et al., 2009), the authors conclude that dependencies on an implementation/architectural level should be separated and handled differently from dependencies on a level of features. We support this conclusion from a different angle by our results. In our study, we showed that specifying dependencies solely on an implementation/architectural level leads to a high chance of missing dependencies on the level of functions.

The extraction of cross-feature dependencies as presented by Cataldo and Herbsleb (Cataldo and Herbsleb, 2011) is comparable to our extraction of feature dependencies. In their study, they showed with statistical significance that the higher the number of feature dependencies is, the higher is the likelihood of integration failures to occur. In our study, we have not investigated the relation between feature dependencies and integration failures but we observed a high number of feature dependencies in real-world multifunctional systems and showed that developers are, in most cases, not aware of them. Together with the results from the study of Cataldo and Herbsleb (2011), this leads to the conclusion that many severe faults are due to feature dependencies.

Technical debt in the context of automotive systems is a relatively new topic. Eliasson et al. recently defined and assessed two types of architecture technical debt for automotive systems: A derivation of the actual system architecture from a previously defined ideal architecture (Eliasson et al., 2015a) and a misplaced component, which is a component that is deployed to a hardware execution unit different from other components that contribute to the realization of the same feature (Eliasson et al., 2015b). Our work adds implicit communal components as an additional type of technical debt relevant in the automotive context. We consider it promising to further identify and investigate types of technical debt specific for a given context (such as automotive systems).

Martini et al. (2014) provide a qualitative model that describes causes of introducing technical debt. One cause they mention is "priority of features over product". They exemplify: *"Small refactorings necessary for the feature are carried out within the feature development by the team, but long-term refactorings, which are needed to develop "architectural features" for future development, are not considered necessary for the release."* (Martini et al., 2014) This matches the situation in automotive companies where the whole development is often organized in feature teams and no team is responsible for an extensible and maintainable system architecture. From our point of view, this is a reason for the large number of implicit communal components we found in the analyzed systems.

A recent trend in automotive architectures is the migration to service-oriented architectures (Kugele et al., 2018; Berger et al., 2017; Sommer et al., 2013) to allow for more flexible and faster development and deployment. In a recent case study, we have shown how an advanced driver assistance function can be migrated to a microservice architecture (Lotz et al., 2019). The definition of modular services may well support our proposed refactoring approach. During the refactoring of an implicit communal component, the component may be migrated to a microservice. Services and service-based definition of software components are also supported in recent automotive architecture standards such as the AUTOSAR Adaptive Platform (Fürst and Bechter, 2016).

8. Summary and conclusions

In this article, we presented an extensive analysis of dependencies between features in automotive software systems. Features are used to structure the functionality that is provided by a system with the goal to decompose the specification. We presented an algorithmic approach to extract feature dependencies from dataflow between components that implement the features.

Our results point to a number of problems that occur in today's development of automotive software systems. Current development processes handle vehicle features more or less as isolated units of functionality. To some extent, this has historical reasons as the automotive industry managed to make their different features as independent as possible such that vehicles could be developed and produced in a modular way. With the rise of software-based features in the vehicle, this independence disappeared (Broy, 2006). The extent and distribution of dependencies between vehicle features challenge the process and methods for requirements specification, system integration and testing (Benz, 2010; Vogelsang et al., 2016).

The results of this study show that dependencies between vehicle features pose a great challenge for the development of automotive software systems. Almost every vehicle feature depends on or influences another vehicle function. We have also seen that describing the dependencies solely on the level of implementing components is insufficient for analyzing them, leading to a 50% chance that a developer is not aware of a specific dependency. Our results empirically underpin the challenges mentioned by Broy (2006); Pretschner et al. (2007), where the authors state that *"functions [of a vehicle] do not stand alone, but exhibit a high dependency on each other so that a vehicle becomes a complex system where all functions act together"*. Dependencies should rather be specified on the levels of features, which demands a clear definition of feature interfaces. We have proposed a dependency-based refactoring approach for components that shows the potential of considering feature dependencies for a more modular system architecture.

Acknowledgments

We thank the employees of our industrial partners for providing the data that we analyzed and the interview participants for contributing to the results. In particular, we thank Steffen Fuhrmann and Stefan Cimander. In addition, we thank Florian Wiesweg for feedback on earlier versions of this manuscript.

References

Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines – Concepts and Implementation. Springer.
Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., Garvin, B., 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In: 5th International Workshop on Feature-Oriented Software Development (FOSD), pp. 1–8. doi:10.1145/2528265.2528267.
Apel, S., Lengauer, C., Möller, B., Kästner, C., 2010. An algebraic foundation for automatic feature-based program synthesis. Sci. Comput. Program. 75 (11), 1022–1047. doi:10.1016/j.scico.2010.02.001.
Batory, D., Sarvela, J., Rauschmayer, A., 2004. Scaling step-wise refinement. IEEE Trans. Softw. Eng. (TSE) 30 (6), 355–371. doi:10.1109/TSE.2004.23.
Benz, S., 2010. Generating Tests for Feature Interaction. Ph.D. thesis. Technische Universität München.
Berger, C., Nguyen, B., Benderius, O., 2017. Containerized development and microservices for self-driving vehicles: experiences & best practices. In: IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 7–12. doi:10.1109/ICSAW.2017.56.
Broy, M., 2006. Challenges in automotive software engineering. In: 28th International Conference on Software Engineering (ICSE), pp. 33–42. doi:10.1145/1134285.1134292.
Broy, M., 2010. Multifunctional software systems: structured modeling and specification of functional requirements. Sci. Comput. Program. 75 (12), 1193–1214. doi:10.1016/j.scico.2010.06.007.
Broy, M., Damm, W., Henkler, S., Pohl, K., Vogelsang, A., Weyer, T., 2012. Introduction to the SPES modeling framework. Model-Based Engineering of Embedded Systems. Springer Berlin Heidelberg doi:10.1007/978-3-642-34614-9_3.
Broy, M., Krüger, I., Pretschner, A., Salzmann, C., 2007. Engineering automotive software. Proc. IEEE 95 (2), 356–373. doi:10.1109/JPROC.2006.888386.
Cafeo, B.B., Cirilo, E., Garcia, A., Dantas, F., Lee, J., 2016. Feature dependencies as change propagators: an exploratory study of software product lines. Inf. Softw. Technol. 69, 37–49. doi:10.1016/j.infsof.2015.08.009.
Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S., 2003. Feature interaction: a critical review and considered forecast. Comput. Netw. 41 (1), 115–141. doi:10.1016/S1389-1286(02)00352-3.
Cataldo, M., Herbsleb, J.D., 2011. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In: 33rd International Conference on Software Engineering (ICSE), pp. 161–170. doi:10.1145/1985793.1985816.
Chen, K., Zhang, W., Zhao, H., Mei, H., 2005. An approach to constructing feature models based on requirements clustering. In: 13th IEEE International Conference on Requirements Engineering (RE), pp. 31–40. doi:10.1109/RE.2005.9.
Crespo, R.G., Carvalho, M., Logrippo, L., 2007. Distributed resolution of feature interactions for internet applications. Comput. Netw. 51 (2), 382–397. doi:10.1016/j.comnet.2006.08.010.
Dajsuren, Y., 2015. On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems. Ph.D. thesis. Department of Mathematics and Computer Science, Technische Universiteit Eindhoven.
Dajsuren, Y., van den Brand, M.G., Serebrenik, A., Roubtsov, S., 2013. Simulink models are also software: modularity assessment. In: 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA), pp. 99–106. doi:10.1145/2465478.2465482.
Donaldson, R., Calder, M., 2012. Modular modelling of signalling pathways and their cross-talk. Theor. Comput. Sci. 456, 30–50. doi:10.1016/j.tcs.2012.07.003.
Eliasson, U., Heldal, R., Pelliccione, P., Lantz, J., 2015. Architecting in the automotive domain: descriptive vs. prescriptive architecture. In: 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 115–118. doi:10.1109/WICSA.2015.18.
Eliasson, U., Martini, A., Kaufmann, R., Odeh, S., 2015. Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: a case study. In: International Workshop on Managing Technical Debt (MTD), pp. 33–40. doi:10.1109/MTD.2015.7332622.
Femmer, H., Vogelsang, A., 2019. Requirements quality is quality in use. IEEE Softw. 36 (3), 83–91. doi:10.1109/MS.2018.110161823.
Fürst, S., Bechter, M., 2016. AUTOSAR for connected and autonomous vehicles: the AUTOSAR adaptive platform. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 215–217. doi:10.1109/DSN-W.2016.24.
Holten, D., 2006. Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. IEEE Trans. Vis. Comput. Graph. 12 (5), 741–748. doi:10.1109/TVCG.2006.147.
Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H., 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (Eds.), Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, pp. 151–165.
Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., 1990. Feature-oriented domain analysis (FODA) feasibility study. Technical Report, CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University.
Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: 30th International Conference on Software Engineering (ICSE), pp. 311–320. doi:10.1145/1368088.1368131.
Kästner, C., Apel, S., ur Rahman, S.S., Rosenmüller, M., Batory, D., Saake, G., 2009. On the impact of the optional feature problem: Analysis and case studies. In: 13th International Software Product Line Conference (SPLC), pp. 181–190.
Kugele, S., Hettler, D., Peter, J., 2018. Data-centric communication and containerization for future automotive software architectures. In: IEEE International Conference on Software Architecture (ICSA), pp. 65–74. doi:10.1109/ICSA.2018.00016.
Liu, J., Batory, D., Lengauer, C., 2006. Feature oriented refactoring of legacy applications. In: 28th International Conference on Software Engineering (ICSE), pp. 112–121. doi:10.1145/1134285.1134303.
Lotz, J., Vogelsang, A., Benderius, O., Berger, C., 2019. Microservice architectures for advanced driver assistance systems: a case-study. In: IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 45–52. doi:10.1109/ICSA-C.2019.00016.
Martini, A., Bosch, J., Chaudron, M., 2014. Architecture technical debt: understanding causes and a qualitative model. In: 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 85–92. doi:10.1109/SEAA.2014.65.
Pretschner, A., Broy, M., Krüger, I.H., Stauner, T., 2007. Software engineering for automotive systems: a roadmap. In: Future of Software Engineering. IEEE Computer Society, pp. 55–71. doi:10.1109/FOSE.2007.22.
Ribeiro, M., Queiroz, F., Borba, P., Tolêdo, T., Brabrand, C., Soares, S., 2011. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In: 10th ACM International Conference on Generative Programming and Component Engineering (GPCE), pp. 23–32. doi:10.1145/2047862.2047868.
Robson, C., 2002. Real World Research: A Resource for Social Scientists and Practitioner-Researchers. Wiley-Blackwell.
Schätz, B., 2008. Modular functional descriptions. Electron. Notes Theor. Comput. Sci. 215, 23–38. doi:10.1016/j.entcs.2008.06.019.
Shaker, P., Atlee, J., Wang, S., 2012. A feature-oriented requirements modelling language. In: IEEE International Conference on Requirements Engineering (RE), pp. 151–160. doi:10.1109/RE.2012.6345799.
Sommer, S., Camek, A., Becker, K., Buckl, C., Zirkler, A., Fiege, L., Armbruster, M., Spiegelberg, G., Knoll, A., 2013. RACE: a centralized platform computer based architecture for automotive applications. In: IEEE International Electric Vehicle Conference (IEVC), pp. 1–6. doi:10.1109/IEVC.2013.6681152.
Timmermans, B., Cleeremans, A., 2015. How can we measure awareness? An overview of current methods. Behavioural methods in consciousness research, Oxford University Press Oxford, pp. 21–46. 10.1093/acprof:oso/9780199688890.003.0003.
Vogelsang, A., Femmer, H., Junker, M., 2016. Characterizing implicit communal components as technical debt in automotive software systems. In: 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 31–40. doi:10.1109/WICSA.2016.28.
Vogelsang, A., Fuhrmann, S., 2013. Why feature dependencies challenge the requirements engineering of automotive systems: an empirical study. In: 21st IEEE International Requirements Engineering Conference (RE) doi:10.1109/RE.2013.6636728.
Vogelsang, A., Teuchert, S., Girard, J., 2012. Extent and characteristics of dependencies between vehicle functions in automotive software systems. 4th International Workshop on Modeling in Software Engineering (MISE) doi:10.1109/MISE.2012.6226020.
Weiss, M., Esfandiari, B., Luo, Y., 2005. Towards a classification of web service feature interactions. In: Benatallah, B., Casati, F., Traverso, P. (Eds.), Service-Oriented Computing (ICSOC). Springer Berlin Heidelberg, pp. 101–114.
Zave, P., 1999. FAQ sheet on feature interaction. http://www.research.att.com/~pamela/faq.html.

Andreas Vogelsang is an assistant professor (junior professor) for software engineering at the Berlin Institute of Technology (TUBerlin). He is leading the software engineering group at the Daimler Center for Automotive IT Innovations (DCAITI). He received a Ph.D. from the Technical University of Munich. His research interests comprise requirements engineering, model-based systems engineering, and software architectures for embedded systems. He has published his research in international journals and conferences such as IEEE Software, ICSE, and RE. In 2018, he was appointed as Junior-Fellow of the German Society for Informatics (GI). Further information can be obtained from https://www.aset.tu-berlin.de.