Characterizing Implicit Communal Components as Technical Debt in Automotive Software Systems

Andreas Vogelsang, Henning Femmer, Maximilian Junker
Technische Universität München, Germany
{vogelsan,femmer,junkerm}@in.tum.de

Abstract—Automotive software systems are often characterized by a set of features that are implemented through a network of communicating components. It is common practice to implement or adapt features by an ad hoc (re)use of signals that originate from components of another feature. Thereby, over time some components become so-called implicit communal components. These components increase the necessary efforts for several development activities because they introduce feature dependencies. Refactoring implicit communal components reduces these efforts but also costs refactoring effort.

In this paper, we provide empirical evidence that implicit communal components exist in industrial automotive systems. For two cases, we show that less than 10% of the components are responsible for more than 90% of the feature dependencies. Secondly, we propose a refactoring approach for implicit communal components, which makes them explicit by moving them to a dedicated platform component layer. Finally, we characterize implicit communal components as technical debt, which is a metaphor for suboptimal solutions having short-term benefits but causing a long-term negative impact. With this metaphor, we describe the trade-off between accepting the negative effects of implicit communal components and spending the necessary refactoring costs.

Index Terms—Feature dependencies, feature interaction, technical debt, automotive software, empirical research

I. INTRODUCTION

Software development, as performed today in automotive companies, is strongly influenced by existing legacy systems, organizational constraints, and complex OEM/supplier relationships [1]. 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, short-term goals, such as the delivery of a feature, frequently trump long-term objectives like maintainability or extensibility.

The development of automotive features is often characterized by a set of features that are implemented through a network of communicating components. Technically, a component is deployed to a hardware execution unit, which is connected to one or more bus systems that provide signals from all kinds of 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. In many cases, the signals a developer (re)uses originate from the implementation of another feature, however, the developer is usually not aware of that [2].

If we abstract from the bus systems and hardware execution units, we can take a logical view to make the connections of components via signals explicit. Fig. 1 shows a schematic model of such a logical view. In this model, two components are connected by an arrow if one component reads a signal from a bus system that is written by the other component (C1 → C2 means C2 reads an output signal of C1).

Fig. 1. Logical view onto an automotive architecture. Components are connected if one component reads a signal from a bus system that is written by the other component (C1 → C2 means C2 reads an output signal of C1).

If we abstract from the bus systems and hardware execution units, we can take a logical view to make the connections of components via signals explicit. Fig. 1 shows a schematic model of such a logical view. In this model, two components are connected by an arrow if one component reads a signal from a bus system that is written by the other component. In this example, three features (F1, F2, F3) are implemented through a network of communicating components (C1, . . . , C7). Components C8 and C9 are not associated with a specific feature but with a dedicated platform component layer (PCL). The PCL bundles components that implement common functionality for a number of features, e.g. providing sensor values or controlling actuators. Therefore, we call such components communal. When a developer adapts a feature by carelessly (re)using signals from the bus system that originate from another feature, components associated with that feature become, de facto, communal as well. In the figure, component C4 is an example for such a case because it provides signals not only for the containing feature F2 but also for features F1 and F3. When a communal component is associated with a feature, we call it an implicit communal component, whereas we call a communal component explicit when it is associated with the platform
component layer. The difference is that for implicit communal components, reuse is performed in an ad hoc manner, whereas explicit communal components are designed to support reuse. Ad hoc reuse bears the risk that other developers may not be aware of the reuse and the components of the reused signals may not be created for this purpose.

Ad hoc signal reuse that results in implicit communal components introduces feature dependencies (e.g., between \( F_2 \) and \( F_3 \)). In former studies, we have shown that such feature dependencies are numerous in current automotive systems, yet to a large extent unknown to the developers [2], [3]. In this paper, we will additionally show that the feature dependencies are not equally distributed over the components, we will provide a possible refactoring approach, and discuss when refactoring feature dependencies may be reasonable. If a feature contains communal components, several development activities become more expensive. For example, testing a feature in isolation is not possible anymore and changing a signal might break the implementation of depending features. A possibility to avoid these costs is to refactor implicit communal components by shifting them from a feature into the PCL. Such refactorings, however, are also costly.

Finding a reasonable trade-off between increased development costs that arise from implicit communal components on the one hand and refactoring costs for making them explicit on the other hand calls for an informed decision-making process.

This paper makes three contributions for this research objective. First, we provide empirical evidence that implicit communal components exist in industrial automotive systems. For two systems, we show that 36% resp. 46% of all components are implicit communal components. We furthermore show that feature dependencies are not equally distributed over components. In the examined systems, less than 10% of all components are responsible for more than 90% of the feature dependencies. Secondly, we propose a refactoring approach for implicit communal components, which makes them explicit in a dedicated platform component layer. Finally, we characterize implicit communal components as technical debt, which is a metaphor for suboptimal solutions having short-term benefits but causing a long-term negative impact. This characterization includes a list of cost factors that describe additional efforts for features that contain implicit communal components and a list of cost factors that describe efforts necessary for refactoring these implicit communal components.

II. BACKGROUND, TERMS, AND DEFINITIONS

First, we introduce how automotive systems are structured by features and components, then, we introduce the notion of communal components and, finally, we provide details on the technical debt metaphor.

A. Feature-oriented Development of Automotive Software

An automotive software system and its development is often structured according to features that capture user-visible behavior such as airbag, cruise control, or start-stop system [4]. The different features are implemented by a complex network of components that communicate via signals transmitted over bus systems that are open to all components connected to that bus system. Components are functional blocks that provide behavior ranging from sensor handling to signal processing and actuator triggering. Automotive companies try to keep features as independent as possible from each other because they usually divide 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 [1]. 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. We use the following definition of feature dependency within this paper (adapted from [2]):

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 \( C_2 \) reads a signal that originates from \( C_4 \), which is associated with \( F_2 \)) and likewise \( F_3 \) depends on \( F_2 \).

In recent studies, we found that, in the studied systems, not only almost every feature depended on another feature, we have also seen that there is a 50% chance that a developer is not aware of a specific feature dependency [2], [3].

B. Communal Components

Definition 2 (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.

We distinguish between implicit and explicit communal components depending on whether a component is associated with a feature or with the platform component layer.

Definition 3 (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. 1, \( C_2, C_4, C_5, C_6, \) and \( C_7 \) are implicit communal components.

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

Definition 4 (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 this dedicated platform layer is to bundle components that implement functionality important for a number of features. This may include components that provide some general signals (e.g., vehicle speed) but also components that collect and process signals for one specific actuator (e.g., different brake demands). In Fig. 1, $C_8$ and $C_9$ are explicit communal components.

C. Technical Debt

Technical debt (TD) is a metaphor reflecting technical compromises that can yield short-term benefits but may hurt the long-term health of a software system. This metaphor was initially concerned with software implementation (i.e., at code level), but it has been gradually extended to software architecture, detailed design, and even documentation, requirements, and testing [5].

Tom et al. [6] have explored the TD metaphor and outlined a first framework. According to the Technical Debt theoretical framework, TD is composed of the following elements: the suboptimal solution (technical debt item), the cost of refactoring it (principal) and the impact (interest) on some quality attributes, for example on maintainability or performance. Based on the information about these three components, software developers, architects, and managers would be able to decide if a TD item is a good or bad investment (for example if the interest paid is more than the cost of refactoring), in order to allocate resources for refactoring. Part of the overall TD is related to suboptimal architectural decisions. This is regarded as Architecture Technical Debt (ATD). The ATD concepts described in [6] are going to frame our study results.

III. STUDY DESIGN

In this study, we identify and characterize implicit communal components in automotive software architectures as ATD item (architectural violations) and aim at exploring the impact (interest) in terms of additional development efforts. We additionally propose a refactoring operation for implicit communal components and explore the necessary effort (principal).

A. Research Objective

Our research goal is to assess the extent of implicit communal components within industrial automotive systems and explore a characterization as technical debt. For this purpose, we follow three research questions.

B. Research Questions

RQ1: How many implicit communal components exist in industrial automotive systems? We want to assess the extent of implicit communal components in industrial automotive systems. The answer to this question indicates the relevance of considering implicit communal components as technical debt.

RQ2: What is the distribution of feature dependencies over implicit communal components? We want to understand how single communal components contribute to the number of feature dependencies within the whole system. The answer to this question may or may not justify an individual assessment of components with respect to their impact on feature dependencies. If feature dependencies are equally distributed over all components, the need for a decision support which components to refactor may not be as relevant.

RQ3: What is the potential distribution impact, or interest, of implicit communal components? We want to identify and characterize the consequences of implicit communal components with respect to development effort. We especially aim at the consequences caused by the feature dependencies resulting from implicit communal components.

RQ4: What is the potential refactoring effort, or principal, of making implicit communal components explicit? With this RQ, we want to identify and characterize the effort necessary to remove an implicit communal component. We especially consider shifting the implicit communal component into a dedicated platform component layer as refactoring operation.

C. Refactoring Approach

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. 2 shows a refactored version of the architecture of Fig. 1, where the implicit communal component $C_4$ is made explicit by shifting it to the PCL. Making implicit communal components explicit reduces feature dependencies. For example, by the refactoring shown in Fig. 2, 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.

In the remainder of this paper, we will show that this change has an impact on efforts for several development activities.
After the calculation of the initial values for the number of components, we implemented and executed an automated analysis. For the first analyzed system, which describes the component architecture of a small truck, the system comprises 57 fully specified features that are implemented by an overall of 269 components. Components may be used to realize more than one feature. The second system describes the component architecture of the driving dynamics and driver assistance domain of an SUV. The system comprises 94 features with 380 components in total. Table I summarizes the characteristics of the two systems. While the first system was developed for a year and was still under development at the time of the study, the second system was already maintained for approximately two years including a number of releases with several feature evolution steps.

### E. Data Collection and Analysis

To answer the research questions, we used two different techniques, which will be described in the following.

1) Automated Analysis for RQ1 and RQ2: To answer RQ1, we extended the analysis with an iterative refactoring operation that, in one iteration, removes 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. After the calculation of the initial values for the number of implicit communal components and feature dependencies, the analysis performs the mentioned refactoring operation and again calculates the remaining number of implicit communal components and feature dependencies. The analysis repeats this step until 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 were the only dependencies a communal component contributed to). The result of this analysis is a decreasing number of feature dependencies for each refactoring of an implicit communal component.

2) Stakeholder Interviews for RQ3 and RQ4: To answer RQ3 and RQ4, we conducted semi-structured interviews as a data generation method. Each interview took around one hour.

The interviews began with an introduction of the purpose of the study and an assurances of confidentiality of the information. This was followed by a presentation of the feature dependency analysis and the results for the two analyzed systems. The interviewees were allowed to ask questions for clarification. Afterwards, we asked the interviewees what kind of extra work needs to be performed in their specific context if a feature has feature dependencies. We did not focus on specific activities in the interviews. We rather asked the interviewees to broadly think of any activity in which they are involved and asked them whether a feature dependency causes extra work for that activity. After this, we introduced our proposed refactoring approach to remove a feature dependency by moving the corresponding implicit communal component into a dedicated platform component layer. In the final phase of the interview, we asked the interviewees what kinds of work is required in their specific context to move an implicit communal component into a dedicated platform component layer.

During the interviews, we collected field notes and recorded audio. We followed a grounded theory methodology [7]. Three researchers coded the interview data independently. In a workshop, we consolidated the codes to assemble a conceptual model of cost factors resulting from implicit communal components. Afterwards, we identified instances of this conceptual model to explore the interest costs of implicit communal components (RQ3) and the principal costs of making implicit communal components explicit (RQ4).

We interviewed three practitioners from the companies of the examined systems (see Table II). Participant P1 is a manager with over 5 years of experience in managing automotive component architectures. P2 and P3 are developers with over 5 years of experience. Participant P3 is responsible for the development and maintenance of a software component that provides consolidated signals to be used in different features. This component closely resembles our concept of the PCL. P1 and P2 were involved in the development of the analyzed systems and were already familiar with the results of the feature dependency analysis that we used for RQ1 and RQ2. We selected the participants based on their availability and experience and with the goal to explore a large spectrum of cost factors that may influence the refactoring decision. The authors and the interview participants already worked together in a number of projects over the last 5 years. Thereby, a common

### Table I

**Overview of the Study Objects and Study Results for RQ1**

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>System 1</th>
<th>System 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Small truck</td>
<td>SUV</td>
</tr>
<tr>
<td>Features</td>
<td>57</td>
<td>94</td>
</tr>
<tr>
<td>Components</td>
<td>269</td>
<td>380</td>
</tr>
</tbody>
</table>

1 The overall vehicle system comprises 142 features. However, at the time of the analysis, the system was not yet fully specified. Incompletely specified features were excluded from the study.
TABLE II
OVERVIEW OF INTERVIEW PARTICIPANTS FOR RQ3 AND RQ4

<table>
<thead>
<tr>
<th>Participant</th>
<th>Role</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Manager</td>
<td>System 1</td>
</tr>
<tr>
<td>P2</td>
<td>Developer</td>
<td>System 2</td>
</tr>
<tr>
<td>P3</td>
<td>Developer</td>
<td>System 2</td>
</tr>
</tbody>
</table>

TABLE III
OVERVIEW OF THE STUDY RESULTS FOR RQ1

<table>
<thead>
<tr>
<th>Results for RQ1:</th>
<th>System 1</th>
<th>System 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit communal components</td>
<td>97 (36%)</td>
<td>175 (46%)</td>
</tr>
<tr>
<td>Feature dependencies</td>
<td>136</td>
<td>1451</td>
</tr>
</tbody>
</table>

language has been established, which prevents terminological misunderstandings possibly threatening the construct validity.

IV. STUDY RESULTS

A. RQ1: Number of Implicit Communal Components

Table III depicts the results for RQ1. 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.

Interpretation: The goal of RQ1 was to assess the extent of implicit communal components in industrial automotive systems to indicate the relevance of considering implicit communal components as technical debt. The presented results show that in the examined systems more than a third 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.

B. RQ2: Relation Between Feature Dependencies and Implicit Communal Components

Fig. 3 shows the remaining percentage of feature dependencies within the two systems after successively refactoring the component that contributes to the largest number of feature dependencies. The figure shows that for both 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 10% of the components. To completely remove all feature dependencies, we need to refactor 13% of components for System 1 and 19% for System 2.

Interpretation: The goal of RQ2 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 differ 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.

C. Theory of Cost Factors for RQ3 and RQ4

As part of the data analysis process, we inductively derived a conceptual model that emerged from the interview data. We extracted the concepts by identifying patterns in the interview data as suggested by Adolph et al. [7]. Fig. 4 shows the conceptual model. Cost factors are in the center of the model. A cost factor impacts a development activity either in the role of principal costs, i.e., it describes refactoring costs, or in the role of interest costs, i.e., it describes increased costs in the future. A cost factor may be influenced by contextual parameters that increase or decrease the severity of the cost factors.

To answer RQ3 and RQ4, we analyze and represent the interview data as instances of the conceptual model. Fig. 5 shows an example of such an instance. Overall, we extracted four parameters mentioned in the interviews: number of feature dependencies, developer experience, geographic distance, and risk. First, the number of feature dependencies associated with an implicit communal component were mentioned to increase
most cost factors. Second, experienced developers are able to compensate some cost factors. Third, geographical distance between the developers increases some of the cost factors. Last, a few cost factors only represent risks, i.e., the underlying costs may or may not arise in the future. When a cost factor is only a risk, its severity is influenced by the probability that the risk will occur in the future. Table IV summarizes all cost factors that we extracted from the interviews. In the table, the cost factors are related to the development activities that they impact and to the parameters that influence their severity.

D. RQ3: Interest of Implicit Communal Components

The second column of Table IV lists the cost factors that the interviewees mentioned as interest costs of implicit communal components, i.e., increased costs for activities that arise from feature dependencies. In the following, we describe these cost factors in detail and underpin them by statements of the interviewees.

Impact analysis: The goal of an impact analysis is to assess whether the change of a signal or its quality characteristics has a negative impact on some desired behavior.

“If the product management wants to employ a cheaper sensor with a lower update frequency, the requirements engineer needs to assess whether this is still sufficient to fulfill the desired (feature) behavior.”  
(P1)

Impact analysis was mentioned as cost factor in all interviews. The interviewees assessed that for a signal that does not constitute a feature dependency, impact analysis is easy since the signal is specifically defined for the purpose of the corresponding feature. If, however, the affected signal constitutes one or more feature dependencies, the requirements engineer needs to reassess and separate the requirements for each feature to be able to assess the impact for each feature. Oftentimes, feature developers just use signals of other features and do not state their requirements explicitly:

“The requirements for shared signals are not explicitly stated by the signal users.”  
(P2)

“Signals that are available on the BUS systems are used without notifying anyone.”  
(P3)

The amount of additional work for assessing all users of a signal increases with the number of feature dependencies that the signal constitutes. However, one interviewee claimed that when a feature is maintained by one developer for a long time, the additional costs are smaller because the developer, over time, knows all users of a signal. Impact analysis as cost factor is only a risk since an impact analysis might never be necessary for a specific signal.

Safety case for feature: A safety case for a feature includes an argumentation that the feature does not produce a hazardous event. As reported in the interviews of one company, the safety analysis for a feature that depends on signals produced by other features needs to include a safety analysis of parts of the other features. This ensures that the signals produced by the other features do not lead to hazardous behavior. Therefore, the safety analysis for a feature with feature dependencies is more expensive compared with a safety analysis for a feature without dependencies. The costs increase with the number of feature dependencies. Additionally, if a signal is used by more than one feature, the safety analysis for this signal may be done redundantly in each feature:

“For signals that are provided by the PCL, we have to do only one safety case. If signals are reused across feature, they are considered in several safety cases.”  
(P3)

Changed signal maintenance: Changing the implementation of a feature (e.g., due to a change request) bears the risk of breaking the implementation of other features that interact with the changed feature. Thus, changing the implementation of features with feature dependencies causes increased costs due to identifying and adapting depending features. This cost factor constitutes a risk that only creates additional costs if an interacting feature is changed. The interviewees stated that experienced developers can compensate these costs to some degree because they have a profound knowledge about the requirements of the depending features. Therefore, some interviewees assessed this issue as not so problematic but still relevant:

“From time to time something breaks when a signal is changed; but then we usually just call the signal responsible and fix the issue.”  
(P2)

Additionally, this cost factor may not be as serious when developers are geographically close (e.g., they are sitting in the same office).

“Changing signals is less costly when people closely work together and see each other on a day-to-day basis. Coffee breaks provide opportunities to exchange information about changing features”  
(P1)

Feature-based deployment: According to the interviewees, the deployment of automotive systems, i.e., the decision which component is executed on which hardware execution unit, is performed based on features. Eliasson et al. [8] characterize the impact of an inefficient deployment in detail. If two features that interact are deployed to distant hardware execution units within a vehicle, this may lead to increased costs in terms of
efficiency (e.g., increased latency or bus load). However, this is only a risk, which needs to be assessed separately for each feature.

**Test setup for feature tests**: Testing the correct implementation of a feature that interacts with other features creates additional costs for setting up the test environment because the environment needs to incorporate valid behavior for features with which the feature under test interacts. These costs increase with the number of feature dependencies.

**Expert knowledge aggregation**: Collecting, processing, and aggregating data from sensors is a challenging task and requires deep knowledge about the physics and electronics. The same holds for controlling complex actuators. One interviewee claimed that if signals are carelessly reused across features, this expert knowledge is dispersed or, even worse, non-experts are responsible for providing a signal:

“Two-thirds of all signals in the PCL are safety critical. I want experts to handle these signals, not some feature developer.” (P3)

“An improper use of the vehicle speed signal has life threatening consequences.” (P3)

The more feature dependencies an implicit communal component has, the more dispersed is the knowledge that this component relies on. On the other hand, one interviewee pointed out that an experienced feature developer may over time also become an expert for some sensors or actuators that are specifically important for her feature. In this case the interest costs are smaller.

**E. RQ4: Principal of Implicit Communal Components**

The third column of Table IV lists the cost factors that the interviewees mentioned as the principal of implicit communal components, i.e., costs that are saved by not shifting an implicit communal component to the PCL (i.e., making them explicit). In the following, we describe these cost factors in detail.

**Extensive requirements specification**: Since the purpose of the PCL is to provide signals for communal use, all interviewees stated that the components within the PCL need to be specified more extensively than components that are “just” part of a feature implementation. More specifically, one interviewee reported that the requirements specification of a component in the PCL needs to list the properties of the provided signals in detail and relate them to the requirements for each feature that uses the signal. On the other hand, the interviewees emphasized the necessity of this step for all components in the future:

“Sure, it’s work to do the RE extensively when refactoring a signal; but in the other case, we just hope that we don’t have to touch the signal anymore in the future.” (P2)

“In the future, standards like ISO26262 force us to do extensive RE for all component anyway.” (P3)

The interviewees pointed out that the larger the number of features that use a signal, the more work is necessary to relate the properties of the signal to requirements of the corresponding features.

**Safety case for signal**: According to the interviewees, signals provided by the PCL are assumed to be functionally safe, i.e., these signals can be assumed to be correct in the safety analysis for a feature. Therefore, shifting a component from a feature to the PCL requires a safety case for providing the component’s signals. A safety engineer needs to assure that the provided signals are safe and therefore do not need to be challenged in the safety analysis of features that use this signal. As a consequence of this procedure, the signals of the PCL must be assured with respect to the highest safety level of all features using the signal.

**Signal extraction**: Usually, it is not necessary to shift an entire component from a feature to the PCL. It is often enough to extract just one or two signals that are (re)used in several features. Therefore, it is necessary to split a component into one part that will be shifted to the PCL and one part that remains in the feature. One interviewee especially stressed this effort because it becomes even higher when there is a conceptual break between the definition of components and actual implementation units such as C modules. The interviewee reported on such a conceptual break between

<table>
<thead>
<tr>
<th>Activity</th>
<th>Interest</th>
<th>Principal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Engineering</td>
<td>Impact analysis</td>
<td>Extensive requirements specification</td>
</tr>
<tr>
<td>Safety Analysis</td>
<td>Safety case for feature</td>
<td>Safety case for signal</td>
</tr>
<tr>
<td>Development/Maintenance</td>
<td>Changed signal maintenance</td>
<td>Signal extraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code (re)generation</td>
</tr>
<tr>
<td>Deployment</td>
<td>Feature-based deployment</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Test setup for feature tests</td>
<td>Regression test execution</td>
</tr>
<tr>
<td>Organization</td>
<td>Expert knowledge aggregation</td>
<td>Knowledge transfer</td>
</tr>
</tbody>
</table>

Influencing parameters:  
- Number of feature dependencies
- Developer experience
- Geographic distance
- Risk

Table IV: Cost Factors and Influencing Parameters for Interest and Principal Assessment According to Interviewees

---

Note: The table is provided for context and not included in the text. The table lists the cost factors and their influencing parameters.
feature architectures and actual software architectures and added:

“When a feature developer identifies a component that should be refactored, the software architects often have a hard time to cut this specific component out of their actual software architecture because the architecture does not exactly match the components.”

(P3)

**Code (re)generation:** Refactoring operations such as shifting a component to the PCL is usually performed on the level of (architecture) models. Changes in the model demand to (re)generate the implementing code. The resulting costs of this operation depends on the tool chain employed in the company (e.g., whether automatic code generation is employed).

**Regression test execution:** Shifting a component from a feature to the PCL demands to execute regression tests on all levels, i.e., unit, integration, and system tests. The interviewees mentioned this issue as most critical:

“Every change in the architecture is a huge risk. We need to make sure that everything is still working afterwards.” (P2)

**Knowledge transfer:** The interviewees stated that features and the PCL are usually developed and maintained by different persons. This was considered reasonable since feature developers have a different focus and expertise compared with PCL developers whose focus is on providing (high-quality) signals and not computing some system outputs. When a component is shifted from a feature to the PCL, the knowledge that the feature developer has about the component needs to be transferred to the developer of the PCL. These transfer costs may be smaller if the developers are geographically close because this facilitates communication and further inquiries.

V. DISCUSSION

From the presented results, we conclude that implicit communal components constitute technical debt in automotive software systems that is worth to be managed. We draw this conclusion by connecting the major results presented in the last section in the following way: To implement or adapt features, automotive developers (re)use signals that are produced by components of other features. This leads to a large number of implicit communal components (36% and 46% of all components in our cases). When developers do this, they introduce feature dependencies, which, according to our interview participants, increase costs for several activities in the future (e.g., impact analysis, safety cases, testing, deployment). For most of the mentioned activities, their costs depend on the number of feature dependencies for which a communal component is responsible. In the examined systems, the distribution of feature dependencies to communal components resembles a Pareto distribution (long tail), which means that refactoring some communal components has a much higher impact on feature dependencies than refactoring others. Therefore, for some communal components a refactoring saves much more costs compared with others.

Our perception is that this type of technical debt is currently introduced unintentionally (see [9]), which means that most developers and also managers are not aware of the long-term costs of implicit communal components. The presented approach makes this type of debt explicit and therefore paves the way for a prudent management thereof. The cost factors that we presented in the last section may serve as a basis to develop a cost model that indicates for each communal component whether a refactoring of the architecture should be considered. However, based on the presented results, we are not yet sure whether such an operationalization is possible and delivers reasonable quantified assessments. We will investigate this in future work. In the following, we will discuss further implications of our results for academia and industry and describe their limitations and threats to validity.

A. Implications for Academia

Our study provides – as far as we know – the first characterization of technical debt in automotive functional architectures. This type of technical debt is strongly influenced by the particularities of the automotive context. In fact, just from looking at the refactored architecture shown in Fig. 2 it might not be possible to say that this is “better” than the original architecture shown in Fig. 1. It is the knowledge about the different handling of components that are associated with a feature, compared with components that are associated with the PCL, that enables a characterize as technical debt. The few other definitions for technical debt in automotive systems (e.g., [8], [10]) are similarly dependent on the context. We think that it is a promising research direction to further investigate technical debt in the context of automotive software. We expect that approaches targeting generic technical debt (e.g., code smells) can only cover a small part of technical debt in automotive systems. Further research is necessary to gain a comprehensive understanding of technical debt in automotive systems.

A second interesting observation is the diversity of the extracted cost factors. We initially expected that (refactoring) costs would mainly reflect the costs for performing the refactoring operation (i.e., development costs). However, we found that the majority of costs results from follow-up activities such as testing, safety analysis, and organizational changes. This shows the importance of a holistic view when assessing costs of technical debt and its removal.

B. Implications for Industry

In our paper, we show that refactoring implicit communal components may have a positive impact but also comes with refactoring costs. One of the interviewees from a company that maintains a PCL in their systems told us that currently a group of experienced architects decides which components are candidates for the PCL based on their gut feeling and a rough estimation of the number of users of a signal. This shows that the number of feature dependencies resembles the current approach in practice for identifying refactoring recommendations. Our approach may serve as a basis for future discussions about refactoring candidates. Furthermore,
we also work together with automotive companies that do not define a dedicated layer for platform components at all. The results of our paper provide arguments when introducing such a layer might be a reasonable idea.

Besides the reduction of feature dependencies, a platform component layer additionally defines an interface of what the platform is able to provide for all features. The platform component layer may even be reused for a number of vehicles that differ in their features. In a recent paper, we proposed how features can be described based on a definition of platform capabilities [11]. The question how the PCL can be implemented technically leads to current automotive interoperability standards such as AUTOSAR [12]. This standard comprises a set of specifications describing software architecture components and defining their interfaces. More specifically, AUTOSAR specifies a runtime environment that provides a common interface for application software components on a hardware execution unit. Components of the PCL might be implemented as AUTOSAR services, which are available on all hardware execution units.

C. Limitations and Threats to Validity

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.

From a research methodological point of view, the sample size of our study poses a threat to the validity of the results. We answered RQ1 and RQ2 on the basis of examining only two system instances. For RQ3 and RQ4, we conducted interviews with only three practitioners from two companies. Thus, our list of cost factors may not be complete. Also, the list of cost factors depends on the applied development process, which may be different in other companies. However, this threat does not invalidate the general idea of this paper. It might be necessary to extend or refine the list of cost factors for other companies or development processes. Additionally, it might be necessary to include the view of other stakeholders to get a complete view on all influencing cost factors.

Another 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).

D. Related Work

Feature dependencies are closely related to the notion of feature interaction, which is a well-studied phenomenon in technical software systems [13]. Features and their interaction also play an important role in the development and configuration of software product lines [14]. Recent work (e.g., [15]) acknowledges a difference between intended feature interaction and feature interaction that arises from implementations integrated side by side into one system. Kästner et al. [16], for example, examine the optional feature problem, which describes a common mismatch between variability intended in the domain and dependencies in the implementation. The optional feature problem occurs if two (or more) optional features are independent in a domain, but are not independent in their implementation. While our study does not consider software product lines, our results are transferable to the optional feature problem. While two features of a product line may be independent in the domain analysis, a product that contains both features may have an implementation that contains implicit communal components. This problem could also be framed as technical debt. Since the PCL bundles components that are the basis for other features, the PCL itself may be considered as a (base) feature in a software product line environment.

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 [10] 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 [8]. Our work adds implicit communal components as 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. [17] provide a qualitative model that describes causes for 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.” [17]

This pretty much resembles 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.

In the conducted interviews, we found cost factors impacting a number of activities. The idea of assessing quality and cost factors based on development activities is also represented in activity-based quality models (e.g., [18], [19]). Deissenboeck et al. state that “the separation of activities and [artifact] properties facilitates the identification of sound quality criteria and allows to reason about their interdependence” [18]. We consider it an interesting research direction to investigate the integration of the assessed cost factors with activity-based quality models.

In contrast to our recent studies [2], [3], in which we assessed the extent and distribution of feature dependencies only on the level of features, we refined this analysis to the level of
components (implementation units) in this paper, provided a refactoring approach, and detailed the cost factors that need to be considered before performing a refactoring.

VI. CONCLUSIONS & FUTURE WORK

In this paper, we characterized feature dependencies that arise from ad hoc (re)use of signals of other features as technical debt in automotive software systems. We associate this technical debt with components of a feature that also provide signals used in other features; we call these implicit communal components. We have shown that implicit communal components exist in industrial automotive systems and that some of them are responsible for a large number of feature dependencies. We introduced a possible refactoring operation for implicit communal components and characterize a reasonable application as a consideration of cost factors.

The results of this paper encourage us to explore the possibility to quantify the cost factors to provide refactoring recommendations for existing systems. This quantification may be crucial for the success of our approach in practice. One interviewee stated:

“As long as you cannot put any concrete numbers on costs for the future, no manager will accept a refactoring that immediately costs money.” (P2)

Additionally, we want to explore strategies for managing this type of technical debt. It might, for example, be reasonable to refactor communal components together with specific releases, since at this point tests must be executed anyways.

ACKNOWLEDGEMENTS

Please note that this study was not funded by one of the involved companies. We would like to thank the interviewees for their valuable input to our study. Additionally, we thank Veronika Bauer, Stefan Kugele, and the anonymous reviewers for their helpful comments on earlier versions of this work.

REFERENCES


