Developing the Software of Future Cars: A Car DevOps Approach

Marcel Weller, Miles Stötzner, Floriment Klinaku, Steffen Becker
Institute of Software Engineering, Software Quality and Architecture,
University of Stuttgart, Stuttgart, Germany
{firstname}.{lastname}@iste.uni-stuttgart.de

Abstract

The amount of deployed software in cars is increasing. Simultaneously, software is updated in shorter release cycles. As a consequence, car manufacturers face new challenges in the management of software. The project Software-Defined Car (SofDCar) consists of academic and industrial partners addressing the development of foundations for a new software development methodology for future car generations. This methodology demands for new roles and continuous engineering activities during the whole car software's lifecycle. As part of this project, we research several interconnected topics addressing these challenges. We focus on the modeling and orchestration of car fleets, which includes topics such as variant management of car topologies and Over-the-Air updates. Furthermore, the communication between cars and surrounding systems is investigated for use cases such as cooperative overtaking in which cars autonomously overtake each other. Finally, we put all contributions together in a Car DevOps approach.

1 Introduction

Leveraging software is a significant key to success in the automotive industry. Almost all aspects of modern cars are impacted by the use of software. As a consequence, the amount of software and its variants in cars and in the ecosystems of cars (edge or cloud) is constantly rising. In addition, the software in the car and its ecosystem is no longer statically defined at construction time but needs to evolve over the car's whole lifetime.

As a consequence, the management of the complex ecosystem of the car's software is becoming an engineering challenge. In this paper, we highlight three aspects of the overall challenge we are working on. First, we focus on the *safe implementation* of complex car maneuvers in Car2X scenarios such as cooperative overtaking. This requires tight integration of message exchange to coordinate different control loops in the participating cars. Such features involving a single or a few cars in contrast to fleets of cars are implemented by developers of car features. The second focus is on the *flexible deployment and redeployment of software* throughout all layers of the automotive ecosystem and throughout the lifetime of the car. This activity is coordinated by a variability manager role. The third aspect addresses the scalability challenges in various *Over-the-Air (OTA) update scenarios* posed by the fact that all management activities have to be performed for large fleets of cars in resource-constrained networks. This has to be implemented by fleet operators and their runtime scalability experts. Putting together all three contributions results in a *Car DevOps* approach in which the whole software lifecycle and its variants get managed over the whole software lifecycle. For this, we have identified roles and their tasks resulting from the discussed contributions.

In the literature, we can find already proposed solutions for various aspects of the addressed problem. However, it is still lacking a holistic and integrated approach including tailored processes and accompanying developer roles.

To address the raised challenges, we outline in this paper our contributions to the SofDCar project. SofDCar is funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) and tries to provide holistic solutions to the challenges posed by software usage in modern cars. In particular, we propose the use of MechatronicUML models to analyze and implement complex car maneuvers, the extension of TOSCA towards TOSCA4Cars to manage the car's software and its variability and evolution over the car's lifetime, and the use of Palladio scalability prediction models to analyze the scalability of proposed management activities on the scale of fleets of cars.

The paper is structured as follows. Section 2 introduces our motivating scenario used throughout the paper. Section 3 outlines the use of MechatronicUML models for cooperative overtaking. In Section 4, we outline TOSCA4Cars and provide a sketch of how it helps developers to deal with the huge amount of variants in the automotive domain. Section 5 addresses OTA updates and their scalability needs on the level of fleets of cars. In Section 6, we put together all pieces and outline a Car DevOps approach to support all three contributions on the process level before Section 7 concludes the paper.

2 Motivating Scenario

Our motivating scenario consists of two driving cars whereas one car intends to overtake the other one autonomously, as shown in Figure 1. Thereby, when

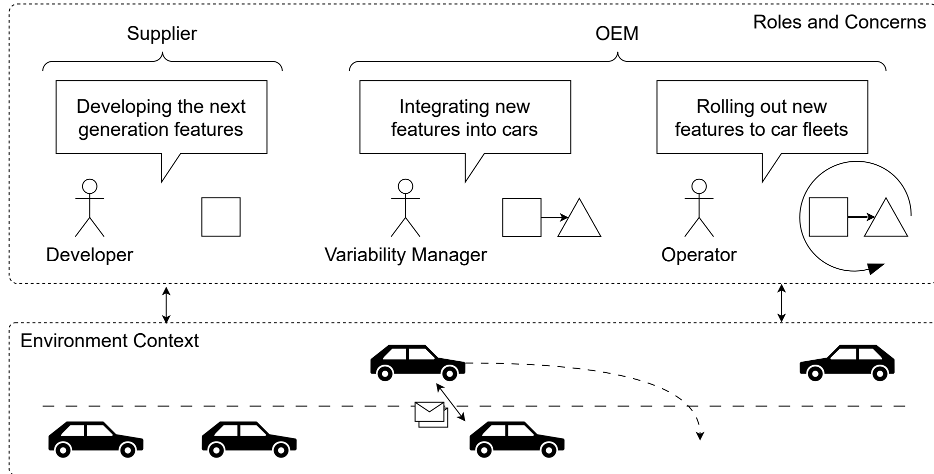

Figure 1: Overview of the roles and their concerns along with the motivating scenario of a cooperative overtaking scenario.

encountering obstacles such as another car, the cars should slow down or even stop. This feature is realized through Car2X communication so that cars contain software components that are responsible for communication with other traffic entities, i.e., other cars or infrastructure like traffic lights. The communication can be carried out either directly between the entities or over a communication service running on the edge. These different components are provided by different suppliers and form a complex system that must be integrated and rolled out by the OEM into the cars. There are different roles involved each having its own specific concerns.

At the side of the supplier, there is the *developer* role. The responsibility of the developer is to design and implement a specific feature of the car. Such a feature can, for example, be the aforementioned autonomous overtaking and therefore include functionality that involves multiple components. The feature is then supplied to the OEM and integrated into the car ecosystem, including, e.g., backend systems in the cloud or edge. The developer wants to provide an implementation of the feature that is functionally correct and fits the design. At the same time, they want to keep the implementation effort low to minimize cost and time. Moreover, during the lifecycle of the cars, the developer is responsible for providing updates to the features, e.g., improvements or security patches.

At the OEM side, the *variability manager* role and the *operator* role are involved. The variability manager is responsible for deciding which software and hardware components are installed in which car and how they are configured. Thereby, there is a huge variability introduced due to market, technical, legal, and customer requirements. As a result, every car is unique on its own. Thus, the variability manager is concerned about deriving the component topology of each car while keeping the management effort low. Moreover, new components as well as new versions or variants of components are developed over time and must be integrated into car lines and cars in the field.

The operator at the OEM who is responsible for rolling out new features to the car fleet has concerns on the overall update process. We focus on the concerns of the operator in the context of over-the-air (OTA) updates, in contrary to classical on-site in the workshop updates. Alongside concerns on the functioning of the update process, for example, that updates are correctly deployed and installed on the car, there exist additional concerns related to the quality of the dissemination. The operator has typical concerns about the qualities of service (QoS) for the update dissemination process expressed through questions like: (1) what is the overall update time for a fleet of cars of certain size?, (2) what is the maximum number of cars that can be updated at the same time?, (3) how cost-efficient is the update process?, (4) does it utilize cloud/edge elasticity for variability in the workload represented by the formed platoons and overall fleet size? Such concerns can be addressed early at the time of designing the dissemination process and supported through QoS-aware interfaces at runtime.

In the following, we further discuss each role in more detail.

3 Model-Driven Development of Event-based Automotive Applications

Cars are distributed cyber-physical systems (CPS) consisting of over 100 electronic control units (ECUs) [8]. The software in a car is not only an integral part of realizing advanced functionality such as autonomous driving functions but is also used for implementing basic functionality like braking. Therefore, car software is highly complex, security- and safety-critical, and often requires hard real-time properties. However, communication based on Remote Procedure Calls (RPC) is used predominantly which results in a tight coupling

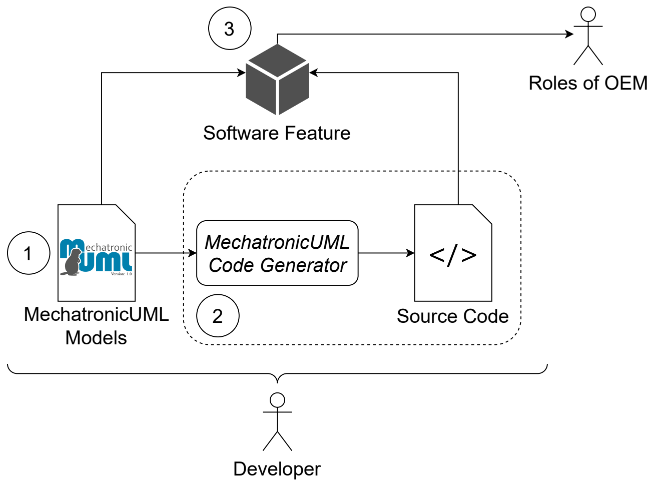

Figure 2: Envisioned workflow of the developer: The developer models the software feature using the MechatronicUML (1), generates the source code with the MechtronicUML Code Generator (2), and supplies both artifacts combined as the software feature to the roles of the OEM (3).

between individual services of the whole car software system. This can lead to a fault in one service propagating through large parts or even the entire system. Additionally, cars are no longer seen as final products but can be updated with new functionality after production, and consequently, car software should be maintainable and able to evolve.

To support *developers* in designing and implementing car software, we apply the MechatronicUML (MUML), a model-driven software engineering (MDSE) method specifically designed for CPSs that accompanies developers throughout the whole development process [2, 4, 15]. The MUML provides a language to model car software independent of the hardware platform. Developers can use the MechatronicUML Code Generator to automatically generate the source code based on the abstract MUML models so that less or even no manual implementation is needed. This reduces the effort for the developers and ensures that the implementation corresponds to the design. Furthermore, the MUML provides means to model behavioral aspects of the software which explicitly considers hard real-time properties and supports event-based communication. Based on this, we investigate the application of event-based communication in and around cars to develop loosely-coupled services that are less error-prone and better to maintain. This includes communication between the software components in a car and Car2X communication.

In the following, we describe how we envision the application of the MUML by the developer role and describe the relevant features of the MUML in more detail. We present an overview of this process in Figure 2. As the first step of this process, the developer designs the software feature by creating models with the MUML (see Step 1 in Figure 2). For this, the MUML follows an approach called the *platform-independent modeling*. It allows the developer to model the software feature as *components* that are independent of the actual vehicle hardware platform. According to the motivating scenario, we create one component for the driving functionality and another one for the overall coordination like communication with other cars. From these components, the developer defines *composite components*, e.g., the whole car. The (composite) components are parameterized and can be instantiated into *component instances* with specific parameter values.

For expressing the interfaces of a component, the developer adds *ports* to the component. The developer then connects the ports of different components and defines and assigns *protocols* to these connections to define the data that is exchanged. Through these protocols, the developer adds Quality of Service (QoS) assumptions to connections for modeling hard real-time properties if required. The developer models the behavior of the defined software components using *state charts* that they associate with a specific port of the component. State charts consist of states and transitions between those states. The transitions are bound to conditions like a specific incoming message that define when the system transitions from one state to another. Based on the motivating scenario, the fast car may send an overtaking request to the slow car, and should then change the lane and accelerate if the slow car accepts. We can model this behavior in the state charts: When the port of the fast car receives an incoming “overtake request accepted”-message, it switches from the default driving state into the overtaking state which triggers internal functions to switch the lane and accelerate.

At a later stage, the developer models the different ECUs in the car or other computing nodes as hardware components in a hardware platform model, including their resource capabilities. After that, the developer creates an allocation of the software components of the platform-independent model to the hardware components. This results in a *platform-dependent* model that comprises all given information. With this approach, the developer can design the car software independent of the target hardware platform, and the car software can potentially be used for different car variants, car series, or even by different OEMs.

Moreover, as shown by Step 2 in Figure 2, the developer supplies these MUML models to the open-source *MechatronicUML Code Generator* for generating a specific implementation [10]. The code generator is implemented in the Eclipse Modeling Framework (EMF) for realizing model-to-model and model-to-text transformations. It already supports the generation of source code in the C programming language and several communication protocols. More importantly, the code generator is extensible so that the developer can add

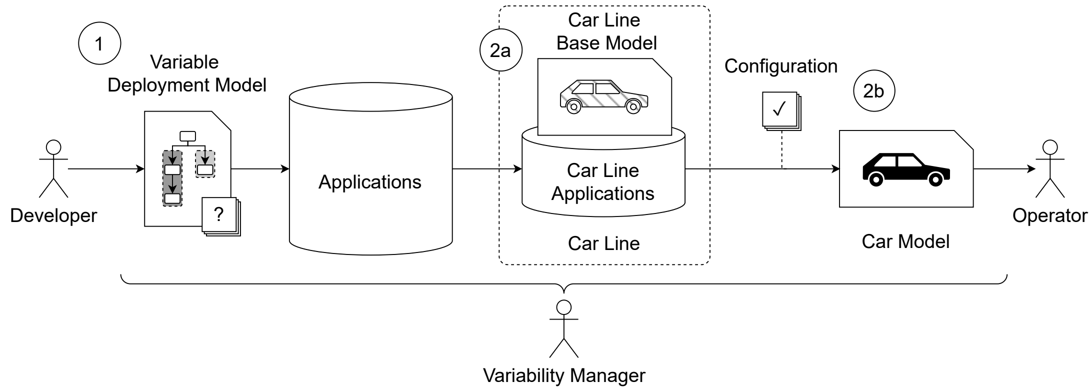

Figure 3: Overview of the method for variability management of automotive deployment models.

transformations for their custom hardware platform or protocols. For example, we added support for the Message Queuing Telemetry Transport (MQTT) protocol to enable event-based communication between the cars. With this, the code generator produces the code for sending and receiving MQTT messages on the corresponding software components. In addition, it generates configuration files for an Eclipse Mosquitto MQTT server, a message broker software that can be deployed on a backend server to which the cars connect.

Finally, the developer supplies the generated source code together with the MechatronicUML models as the software feature to the roles of the OEM (see Step 3 in Figure 2). For providing a new version of the software feature, the developer must only update the MechatronicUML models and regenerate the source code.

4 Variability Management of Automotive Deployment Models

The *variability manager* is responsible for deciding which software and hardware components are installed in which car and how these components are configured. Manually managing software and hardware is error-prone and time-consuming. In the cloud domain, there are established deployment automation technologies, such as Ansible, Puppet, and TOSCA, that could be used to manage software and hardware in cars. Such technologies are typically based on deployment models which declaratively model the desired application state. By interpreting such deployment models, these technologies then automatically derive required deployment and management tasks [7].

However, there is a huge variability in the automotive domain due to technical, legal, and marketing requirements. For example, autonomous driving features, such as the cooperative overtaking application from our motivating scenario, require specific sensors which are not installed in every car and might be allowed in the US but not in Germany. Moreover, cars are highly customized to address customers' preferences or budgets, thus, autonomous driving features might be disabled. As a result, each produced car is unique on its own. Manually creating and maintaining a deployment model for each car is error-prone and time-consuming.

To tackle this, we envision a method to manage applications in the automotive domain based on deployment models in combination with variability management concepts from product line engineering. Product line engineering is an established method to manage the variability of products with the goal to derive a customized product [1, 9]. We chose the *Topology and Orchestration Specification of Cloud Applications (TOSCA)* [12] as the underlying deployment automation technology since TOSCA is an open standard that promises vendor-neutrality and technology-independence. Moreover, we introduce *TOSCA4Cars* as an implementation of our method.

Our envisioned method is based on three building blocks: (1) a metamodel for variable deployment models to model the different variants of how an application can be deployed, (2) a method to derive the deployment model of specific cars by combining variable deployment models, and (3) a method to update the deployment models of specific cars. An overview of our method is given in Figure 3.

The variability manager models the variability of each application in a so-called *variable deployment model* [14] (see Step 1 of Figure 3). A variable deployment model is a deployment model that contains components and relations which have presence conditions assigned which represent, e.g., technical or legal requirements. After variability is resolved, elements are removed whose conditions do not hold. For example, the cooperative overtaking application from our motivating scenario is modeled as such a variable deployment model which contains all required components but also all the different possible hosting components. We discuss this building block in more detail in previous work [14]. As a first prototype, we have developed the open-source TOSCA management

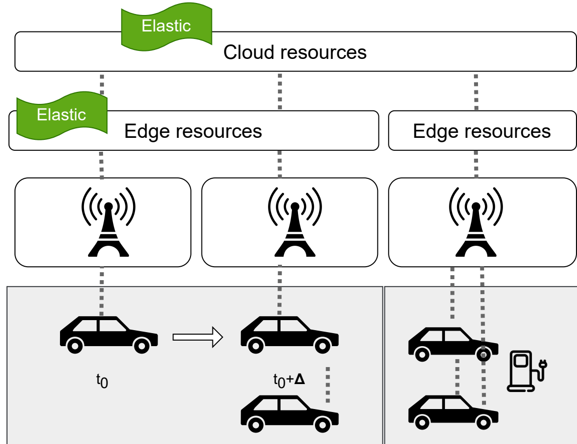

Figure 4: Elastic OTA Updates

and processing tool OpenTOSCA Vintner[1].

To manage not only single applications but complete cars, the variability manager creates for each car line a *car line base model* and a *car line applications set* (see Step 2a of Figure 3). The car line base model is an incomplete deployment model that models, e.g., the architecture of a specific car line in an abstract manner. Moreover, this base model does not only consider applications inside the car but also applications in the cloud or at the edge which are consumed by the applications running in the car, e.g., used to communicate with other cars. The car line applications set is a set of applications that are allowed to be used in the respective car line. The variability manager derives the *car model* of a specific car by combining the car line base model with selected applications from the car line applications set (see Step 2b of Figure 3). Therefore, the variability manager requires a configuration that represents, e.g., legal requirements and customer preferences. Since the car model is a deployment model, it can be executed to deploy and manage the applications inside the car.

To integrate new applications into existing car lines and cars in the field and to update already running applications, the variability manager must update each car model. Therefore, the variability manager conducts the first two steps once again: new variable deployment models are added to the car line applications sets and car models are regenerated. Thereby, already bound variability and limitations must be respected. For example, if a new version of the cooperative overtaking application requires a sensor that is not mandatory for this car line then this application can not be installed in cars that have not been produced with this sensor installed.

5 Elastic Over-the-Air-Update Strategies for Car Fleets

Due to continuous innovation and development, there is a large number of software updates that are rolled out to end-users. This applies also to software that is running in a vehicle. Hence, as we introduce, operators at the OEM side have reliability, security, and performance concerns for the update process. Several off-the-shelf solutions for over-the-air (OTA) updates exist that span across the computing continuum, from edge to cloud data centers (see Figure 4). The focus of our work lies in researching relevant architectural design decisions that balance the involved trade-offs for the update process (i.e., how a desired security level degrades performance).

Adopting off-the-shelf solutions for the over-the-air update process may not satisfy the desired non-functional requirements such as a certain end-to-end completion time or a target utilization of resources. Through understanding the present trade-offs we help engineers develop better solutions for realizing over-the-air updates. Our work is focused on two goals: first, understanding the different types of updates and update scenarios for software in cars. The second goal lies in determining update strategies that meet security and safety requirements and yield a good level of performance at a low cost. The latter goal can be paraphrased as aiming for *elastic OTA updates*.

To better understand the different types of software updates for vehicles, we work on a classification scheme and a set of scenarios with different peculiarities. We consider two main scenarios: (1) OTA update while driving and (2) OTA update at a stop (e.g., home or charging station). Both scenarios can be refined further to create more concrete scenarios which would either benefit from the new conditions or be constrained by them. For example, running an OTA update at a charging station may have a more reliable network.

Another part is the characterization and classification of software updates for cars. For example, Steger et al. [6] classify software updates for vehicles into three different criticality classes: non-critical (mainly entertainment), body and control (e.g., ventilation), and highly critical (e.g., safety-critical functions that impact driving). The size of the update influences performance and end-to-end dissemination time. The classification of software updates for cars in different dimensions helps in choosing fitting update strategies for different classes.

Since realizations for OTA software updates require computational resources from the edge to the cloud, there exist several sources for performance bottlenecks. Moreover, since the demand on each layer may change, the underprovisioning of resources leads to degraded performance. Although the overprovisioning of resources improves performance, it increases cost. Ide-

[1] `https://vintner.opentosca.org`

ally, we can forecast the workload and provision the minimal amount of resources to handle the current demand. Through model-based performance engineering techniques, we aim at understanding better the involved trade-offs and designing OTA update strategies that meet such performance and elasticity requirements. The acquired knowledge can help in decision making and refinement of current state of the art frameworks.

Through a review of existing frameworks and literature, we extract several design decisions that have an impact on the overall performance of the OTA update process. For example, we distinguish three modes of interaction between the update client in the vehicles and the update server: *Polling-Intervals*, where clients regularly, at a predefined rate, poll the server, *Push*, where images are pushed directly to the clients, *Optional*, where clients are notified for an update, but the update is optional and depends on the driver, i.e., the driver can decide when and whether to apply the update.

Since software updates are rolled out and managed for a car fleet, another aspect is the continuity and lifetime of the managed fleet that undergoes an update: in one case, the size of the managed fleet does not change after a rollout is started (*snapshot*), whereas in the other case, vehicles can be added dynamically to the fleet and receive automatically the update (*continuous*). A third design decision concerns the image type: complete image updates are rolled out to cars in one case, whereas in the other case, *delta* updates are created. In *delta* updates, images are created and sent as a *delta* to the currently installed image, so only the changed file blocks are rolled out.

Such design decisions in conjunction with decisions on the runtime policies that govern the elastic provisioning of resources yield the quality of a particular solution. For elasticity prediction, we take a twofold approach. First, we predict the impact and capability of cellular networks like 5G and lower-level communication protocols on the overall performance. For this, we rely on simulation libraries that allow us to simulate Car2X scenarios in a network. One instance is Simu5G [11], that takes an end-to-end approach and allows the simulation of all protocol layers for Car2X scenarios. Second, we estimate the performance at the architectural level including the design of scaling policies for edge and cloud resources. We achieve this goal by performance modeling of the architecture using Palladio [5]. In addition, we evaluate the suitability of resource provisioning policies by modeling them at the architectural level [13]. Through our approach, the operators can predict the performance impact of the employed resource provisioning policies. Moreover, they know the impact and importance of several architectural design decisions and the main performance bottlenecks across the computing continuum.

6 A Car DevOps Approach

So far we have introduced different roles and concerns involved in the development and operation of software for future cars. However, what is still missing is a holistic method that combines these roles into one process. In the following, we discuss how the presented roles interact and work with each other, and how they share information and artifacts. An overview is given in Figure 5.

The developer is responsible for designing and implementing new features. The corresponding developed application and the associated MUML models are passed to the variability manager who then creates a variable deployment model. Different variable deployment models are then combined by the variability manager to derive the car model of a specific car. However, the variability manager does not execute this car model on their own but passes it to the operator who executes the car model using, e.g., a specific strategy to roll out new features in order to level out workload spikes. After the feature has been rolled out to cars in the field, the operator collects new data from in-the-field observations. The developer evaluates these data to improve the feature, develop new features, or fix bugs. The new or updated applications and their MUML models are then once again passed to the variability manager. This essentially restarts the whole process.

However, the process is not as straightforward as just described. Further interactions between the roles are required. For example, who is responsible for generating the code based on MUML models? A platform-specific model is required for the code generation which is, however, only known once the car model has been derived. Thus, the variability manager must interact with the developer once the car model has been derived or must be able to generate the code on their own. It may also be possible that the variability manager already provides one or multiple platform-specific models representing possible combinations of hardware components to the developer, as the OEM has information about the complete car software, including features from different suppliers.

Another aspect is, that the developer has the most expertise considering their own application, thus, knows the most about modeling the corresponding variability. To utilize this, we envision roles with shared responsibilities to bridge expertise in different domains. Roles with shared responsibilities are an already established concept in DevOps [3] in which developers also take responsibility for operating their own applications due to their application-specific expertise. Therefore, we introduce the roles *DevVar*, *VarOps*, and *DevVarOps* which consider not only development and operations but also variability management.

The *DevVar* is a developer who also takes the role of the variability manager when it comes to managing the variability of their own applications. Therefore, they

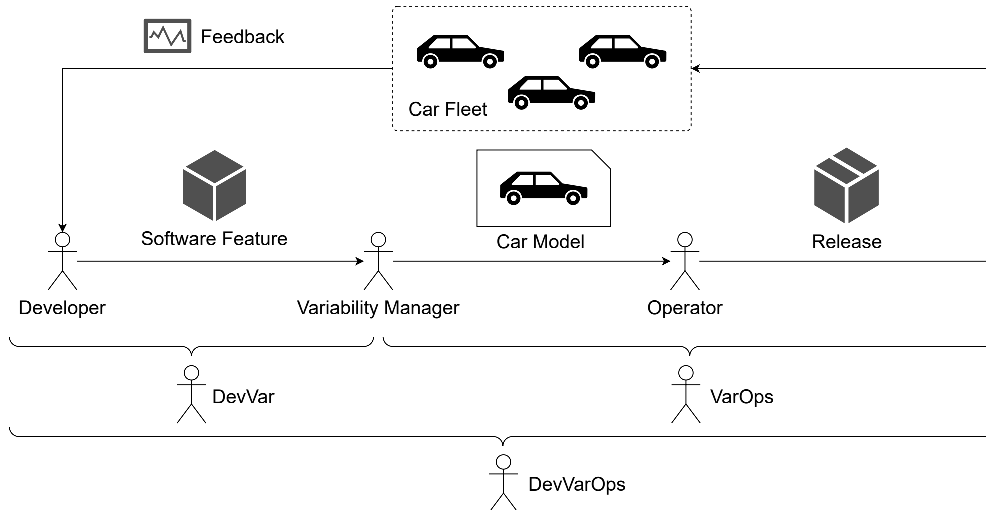

Figure 5: Overview of the different roles involved in our envisioned DevOps approach for cars.

do not only provide MUML models but also variable deployment models for the variability manager at the OEM who then integrates these models into car lines considering OEM-specific requirements.

The *VarOps* is a variability manager who also takes over some responsibilities of the operator. For example, they are responsible for load forecasting, i.e., predicting the load of upcoming updates, based on their knowledge of the existing variability in the fleets.

Ultimately, the *DevVarOps* is the combination of all roles, thus, a developer who takes responsibility for managing the variability of their own applications along with operating them.

Figure 1 shows that the roles can be distributed among different organizations, for example, a supplier employing the developers and an OEM employing the variability managers and operators. Similar to DevOps, we envision that the roles with shared responsibilities presented above are not necessarily executed by a single person, but a team. The team members can be employed by different organizations and work closely together to achieve the common goal of the role with shared responsibilities. Thereby, the team members contribute organization-specific information (e.g., employees of the OEM may contribute details about the hardware platform) while also ensuring the respective objectives of their organization.

7 Conclusion

In this paper, we have outlined three aspects of a future Car DevOps approach. We describe the use of MechatronicUML for coordinated overtaking, TOSCA4Cars as a variability model for car software deployments, and Palladio models for the scalability of OTA updates on the fleet level.

Our contributions help software developers in the car software domain to manage a more flexible development process than it used to be in the past. Developers can work in new roles aiming towards a more flexible and more variant-aware software lifecycle.

In future work, we both need to define the models sketched in the paper but also a process to keep them up-to-date during the lifetime of the software. We will continue to work on case studies to demonstrate the feasibility of our approach in close collaboration with the rest of the SofDCar consortium.

8 Acknowledgements

This publication was partially funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) as part of the Software-Defined Car (SofDCar) project (19S21002).

References

[1] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering.* Springer Berlin Heidelberg, 2005.

[2] T. Stahl, M. Völter, and K. Czarnecki. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, Inc., 2006.

[3] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective.* Addison-Wesley Professional, 2015.

[4] S. Dziwok et al. *The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling.* Tech. rep. tr-ri-16-352. Version 1.0. Zukunftsmeile 1, 33102 Paderborn, Germany: Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Dec. 2016.

[5] R. H. Reussner et al. *Modeling and simulating software architectures: The Palladio approach.* MIT Press, 2016.

[6] M. Steger et al. “SecUp: Secure and Efficient Wireless Software Updates for Vehicles”. In: *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*. Ed. by P. Kitsos. IEEE Computer Society, 2016, pp. 628–636.

[7] C. Endres et al. “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS 2017)*. Xpert Publishing Services, Feb. 2017, pp. 22–27.

[8] M. Staron. *Automotive Software Architectures*. Springer International Publishing, 2017.

[9] K. Pohl and A. Metzger. “Software Product Lines”. In: *The Essence of Software Engineering*. Cham: Springer International Publishing, 2018, pp. 185–201.

[10] U. Pohlmann. “A Model-driven Software Construction Approach for Cyber-physical Systems”. PhD thesis. Universität Paderborn, Heinz Nixdorf Institut, Softwaretechnik, 2018.

[11] G. Nardini et al. “Simu5G–An OMNeT++ Library for End-to-End Performance Evaluation of 5G Networks”. In: *IEEE Access* 8 (2020), pp. 181176–181191.

[12] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS). 2020.

[13] F. Klinaku, A. Hakamian, and S. Becker. “Architecture-based Evaluation of Scaling Policies for Cloud Applications”. In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2021, pp. 151–157.

[14] M. Stötzner et al. “Modeling Different Deployment Variants of a Composite Application in a Single Declarative Deployment Model”. In: *Algorithms* 15.10 (Oct. 2022), p. 382.

[15] Fraunhofer Institute for Mechatronic Systems Design IEM. *MechatronicUML*. Available online: `http://www.mechatronicuml.org` (accessed on May 11th 2023).