A Domain-Specific Language Based Architecture Modeling Approach for Safety Critical Automotive Software Systems

Stefan Schlichthaerle
BMW Group
80788 Munich, Germany
Stefan.Schlichthaerle@bmw.de

Klaus Becker
BMW Group
80788 Munich, Germany
Klaus.Becker@bmw.de

Sebastian Sperber
BMW Group
80788 Munich, Germany
Sebastian.Sperber@bmw.de

***Abstract*—Modeling software architecture for software intensive systems like future automated driving vehicles becomes increasingly complex, compared to Advanced Driver Assistance Systems currently available on the market. The complexity and novelty of the customer functions and the hardware and software platforms demands for an agile development methodology. In this paper, we introduce why and how we at BMW apply a *Treat Architecture like Code* approach for the software development for automated driving. We introduce the requirements which lead us to this approach, the tools and artifact flows around the tools, and how we embed this into our agile development. We show examples from a real vehicle function.**

***Index Terms*—Software architecture, Agile software development, Service-oriented systems engineering**

I. Introduction

In the automotive domain, innovations are to a huge extend driven by software, particularly in the last 15 years [1] [2]. Innovative vehicle features, like Advanced Driver Assistance Systems (ADAS) and future highly, or fully automated driving features (SAE levels 3-5), are mainly based on software. Therefore the amount of software in vehicles has been and will continue to increase dramatically.

Additionally, especially in the context of automated driving, the requirements for the engineering problems cannot be completely predefined in the beginning and then afterwards developed, like in classical development processes. This is because many engineering problems are Volatile, Uncertain, Complex and Ambiguous (often abbreviated with VUCA), raising the demand for agile development models [3]. The VUCA factors influence all stages of engineering, like requirements engineering, software architecture design, software implementation, testing at different integration levels, and also the engineering and integration processes.

Because of the complexity of the development of automated driving, agile software development is required in a large-scale with many software development teams. Therefore BMW adopted a large-scale scrum (LeSS) approach for the teams involved in development of automated driving [4].

In this paper, we present our requirements and our approach for a methodology for software architecture design in a large-scale agile software development process of a Software-Platform for automated driving. The Software-Platform is based on Adaptive AUTOSAR [5]. [1] We also show how Continuous Integration (CI) influences our methodology on describing software architecture.

II. Requirements and Goals for Software Architecture Design

A. Software System and Development Process Overview

The mixed safety-critical software platform for automated driving, for which the architecture modeling approach presented in this paper is used, consists of different CPUs that use Adaptive AUTOSAR as middleware. The communication between the CPUs is enabled by the SOME/IP protocol [6], an automotive standard for Ethernet communication, specified and used as part of AUTOSAR. During the software engineering process, different teams from multiple internal departments and external contracted partners and suppliers contribute software entities into the Continuous Integration (CI) infrastructure. The agile development process is based on the LeSS framework, which enables agile methodologies for big project setups [7].

B. Agile Working Model

A fundamental aspect of agile methodologies is the ability to react fast towards changed requirements [8]. In classical development processes like the waterfall approach, all product development phases follow a well-defined sequence, which starts with requirements engineering and ends with the product release. Changes in this sequence are hard to deploy and are often perceived like a failed project result. In contrast to the waterfall approach, an agile software development process always involves the full cycle to introduce a change, starting from updated requirements, changed architecture, source code and tests. The product development always iterates through this cycle, resulting in small product increments which eventually sum up to the final product.

In an large scale agile software development project with multiple distributed development teams, an essential success

[1] https://www.autosar.org/standards/adaptive-platform

factor is to have a stable basis in a master repository, in which always a certain level of functional quality is ensured. For instance, it shall compile and automated tests on different levels shall be ok. This high quality stable master (sometimes also called *green master*, because all tests are always green) can only be established by avoiding that people directly commit their changes into the master without any quality check. Hence, we setup a staged development process with semi-automated *quality gates*. Developers create small, short living, development branches, do their changes in the branches, and then create a pull request (PR) to request a takeover of their changes into the master. Once the PR has been triggered, two stages of automated quality gates are triggered.

1) check
2) gate

To pass the *check*, at least one manual review has to be done by another developer, which either declines with comments requesting improvements or fixes, or approves the changes. Only when the review and all automated tests done during the *check* are ok, the *gate* is triggered and performs additional deeper tests based on a virtual merge with the master (and with all changes of the master that have been added after the development branch has been created). If the automated tests of the *gate* are also ok, the change will be merged into the master, see Fig. 1.

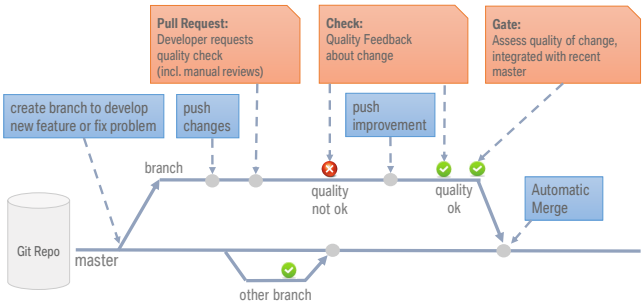

Fig. 1. CI Integration process with quality gates and automatic merging

But what does this staged continuous integration process mean for the development of the software architecture?

C. Treat Architecture like Code

A key enabler for an agile development process is the ability to change all parts of the product in an agile way. Source code repositories and scalable CI infrastructure enable an agile development of source code. But other parts, like architecture models or on-board network descriptions, which are also crucial building blocks of the overall product, are often still treated in a different way, preventing fast iteration cycles. Reasons for this are legacy tools, or a development process which emphasizes on big releases on fixed dates, instead of fast and small incremental updates. To address this challenge, the architecture description shall be kept as close to the code as possible, to avoid architecture erosion by deviations between architecture and code. This leads us to the requirement that ideally, the architecture description is treated in the same way like the source code, homogeneously in the same CI infrastructure. A textual architecture description is best suited for this, as automated merges can be done for the architecture description in exactly the same way as for source code.

This is why we decided to choose a textual *Domain Specific Language (DSL)* like *Franca+* to describe those parts of the architecture from which code is going to be generated (see section V). The architecture model is stored in text files within the same repository as the corresponding source code, accompanied by a tool chain running in the CI infrastructure in the cloud. Thereby we ensure that always, with each delivery, the architecture models match the code and to the build system configuration files. Hence, we integrate the development of architectures with *Franca+* and the development of software code into the same CI process.

III. Related Work

Agile methodologies and development workflows are widely used within software engineering among different industries. A key aspect of software development is the related architectural design. In this section, we briefly describe related work in agile software architecture development.

In [9], the author talks about the alignment of architecture work with agile teams. Based on his experience as software architect, there are often difficulties when software architects and agile development teams work together, because their priorities and scope may vary. For example architects tend to work on a comprehensive design upfront, which shall cover all aspects, whereas agile teams following the agile manifest work continuously on their product and regularly deal with changed requirements, which may impact their overall product.

To address these differences, the author derives a set of architecture practices, which helps software architects and agile teams to collaborate and benefit from each other. Among others, he emphasizes that sharing information over simple tools is key to bring both parties together. Additionally, he describes that incremental deliveries and *Deliver something that runs* are key to match with the development workflow of the agile team. The author does not introduce concrete technical solutions to enable these practices in a project setup, though. In our project, these architecture practices have been clearly identified as well and we address them with our DSL-based approach for architecture modeling.

The work in [10] introduces research questions and study results about the requirements to apply Service Oriented Architectures (SOA) in safety-critical automotive software. For the modeling of service architectures, they propose to use techniques such as UML or SoaML. SoaML enables formal modeling of service interfaces on syntactic and semantic level. They propose to develop service interfaces with service description languages. Typically those service description languages are also text-based, like WSDL. The presented study result was

that the vast majority of interviewed persons agreed that a model based approach for SOA development is needed, and that services should be designed independently from a target hardware platform. The *Franca+* approach which we apply supports this. The textual modeling approach with *Franca+* is motivated by the *Treat Architecture like Code* idea to enable the homogeneous handling of architecture and code in the same CI process, to avoid deviations between architecture and code. As service middleware, we use SOME/IP in our project. Due to the usage of platform-independent models and the platform-specific deployment models (see section IV), different service middleware could be used, like for instance also *Data Distribution Service (DDS)* [11], which can be applied as alternative to SOME/IP in the AUTOSAR Adaptive Platform. [2]

IV. Modeling Approach

In section II-C, we described the philosophy of *Treat Architecture like Code* and the expected benefits and improvements for product development in our domain. To transfer this overall approach into an automotive software project, we developed a modeling approach which is based on latest tools and processes for agile software development and fulfills all requirements necessary for the development of safety critical systems. The current status of our tool chain's qualification regarding these requirements is discussed in V-D.

In this section, we will first describe the key entities of our model and their representation. Afterwards, we introduce the language which is used to describe the entities.

A. Model Fundamentals

Our model consists of two horizontal layers.

- The *model layer* contains platform-independent definition of interfaces, data types and components.
- The *deployment layer* maps the platform-independent model to concrete technologies and instances on CPUs available in the system. This layer also contains dedicated properties and settings, which are required during code generation or at runtime of the system.

A component defined on the *model layer* may include provided and required communication ports, which are typed by the used Interface. An Interface itself can contain an arbitrary number of methods, events or fields, which are typed by the data types.

For every entity on the *model layer*, a deployment is added on the *deployment layer*, which adds properties towards the technical realization on the concrete platform. For example an technological independent interface defined on the *model layer* can be extended with properties required for SOME/IP deployment on the *deployment layer*, so that the interface can be used for SOME/IP communication in the vehicle network. With this layered approach, the same software component and interface description on *model layer* can be mapped to different platforms by using different deployment models.

[2]https://www.autosar.org/standards/adaptive-platform

Besides the layered view described, a different view of the model starts from the individual component and introduces a vertical column spanning from *model layer* on top to the *deployment layer* at the bottom. This view expresses the technological-independent model of a component, its interfaces and data types together with a technology-dependent deployment on a concrete CPU. At the same time, the vertical column reduces dependencies to the shared usage of interfaces and data types, which is a crucial prerequisite for the design of a performant tool chain, as we will see in section (V-A).

To express all entities and to store them in text files within a repository, the *Franca IDL*, which is part of the *Franca* framework [12], is used. In the following section, the *Franca IDL* and the extensions introduced in our project are described.

B. Franca

Franca is a framework for defining and transforming software interfaces. The core of it is the *Franca IDL (Interface Definition Language)*, which is a textual language for specification of APIs [12]. *Franca* is a domain-specific language based on the *Eclipse Xtext* framework [13]. Due to its Xtext nature, *Franca* already provides an editing tool with built-in basic validations and a software development kit (SDK) to work with *Franca* files in other tools, at almost no cost. Within our project, we use the *Franca IDL* for defining entities on the *model layer* and an extension of it called *Franca+*, which we use to define software components and their deployment on the *deployment layer*.

1) Franca IDL: *Franca IDL* is an interface definition language which allows defining interfaces, and corresponding data types. It is part of the *Franca* framework, which is published in [12] and described in detail in [14]. As we use most of *Franca IDL* without any changes, except its concept of service instances as service deployments, we won't describe the language features in detail here and refer to the sources available. Instead we emphasize on the extensions we introduced towards the definition of software components, ECUs and their deployment in the next section.

2) Franca+: *Franca+* was developed at BMW to enable a language-based modeling approach for AUTOSAR environments, where software architecture is described in terms of software components and communication via ports. As an extension of *Franca IDL*, *Franca+* reuses Franca's definitions of interfaces and datatypes. A software component's port is an instantiation of such an interface. The above mentioned model is split into two layers and is also applied to *Franca+*, which consists of two domain specific languages, the *Franca Component Description Language* (FCDL) and the *Franca Component Deployment Language* (CDEPL). The *Component Description Language* allows defining software components with ports that provide or require services. The service's content is described in terms of *Franca* interfaces. Furthermore in *Franca+*, model elements called *devices* can be used to model CPUs and their network interfaces. In case that the communication relations between software components shall

be stated in the model view already, connectors can be used to connect the corresponding component ports.

Within the *Component Deployment Language* it is defined which of these new model elements act as deployment targets and how these deployments have to be specified. The actual deployment properties that can be applied for each type are defined in so called deployment specification files, a mechanism reused from *Franca*. Using this approach deployment properties can be edited later on by simply adding them to the specifications without the need of changing the actual tool. To enable our tool chain we introduced deployment specifications to describe SOME/IP and IPC communication, as well as runtime parameters of Adaptive AUTOSAR applications and diagnostic communication. In *Franca+*, service instances are defined as the deployment of service components, with all corresponding deployment information, e.g. for their communication ports.

In this section, we first introduced the fundamentals of our modeling approach, which supports modeling of platform-independent and platform-dependent entities on different layers. Additionally, we described the domain-specific language that is used as modeling language and the extensions for component description developed at BMW.

In the next section, we describe the tool chain used to translate the model towards the platform and to connect the software architecture model to other tools applied in our software and systems engineering.

V. Tool Chain

To embed the described architecture model into the software development process and fulfill the requirements regarding working mode as stated in section II, a comprehensive tool chain is required.

The tool chain consists of two major parts influencing the actual target code generation and one additional part, which is used for documentation purposes.

1) First, the model stored in the repository has to be translated into executable code, which is eventually running on the target hardware in the vehicle. Besides translation into different languages, this part of the chain also contains validation steps, to ensure that errors in the model are discovered as early as possible.
2) Second, there is a part of the tool chain which produces *Franca+* model artifacts mainly from existing data sources like on-board network or diagnosis description.

Figure 2 shows an overview of the parts of the tool chain, contributing to and starting from the *Franca+* model respectively. In addition there is another part that generates *Unified Modeling Language (UML)* elements for graphical modeling tools. In the following sections, the parts of the tool chain are described in more detail.

A. Code Generation towards the platform

The component model is described with *Franca+* and stored in text files within a git repository. Whenever a developer

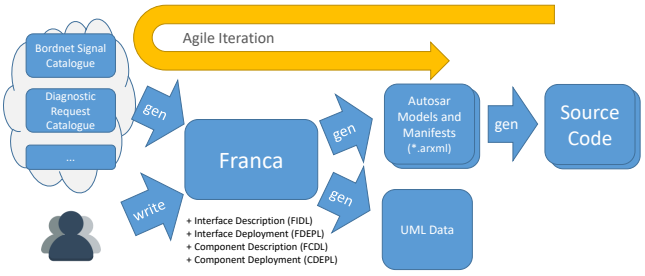

Fig. 2. Franca Toolchain consisting of two parts: Converting existing data sources into the Franca+ model on the left side and translation towards AUTOSAR Models and source code on the right side.

introduces a change request as a pull request, the tool chain is started and tries to compile the change together with the rest of the model.

The first part of the tool chain is a python-based compiler, which reads the *Franca+* model affected by the pull request and translates it into an Adaptive AUTOSAR compatible XML representation. In subsequent steps, the Adaptive AUTOSAR model is further translated into C++ source code and binaries, which are then executed on the hardware. Whenever an error occurs during execution of these steps, no matter if it is related to validation, quality assurance or regression in test cases, the pull request is rejected, resulting in a negative feedback to the developer. To integrate the change, the developer has to first resolve the problems discovered.

In section IV-A, the importance of dependencies between parts of the model has been discussed already. To design a performant tool chain that scales to huge projects, it is very important that a run triggered by a pull request only translates these parts of the model that are directly affected by the change. It shall not be required to consider the whole model to check a partial change.

In a first development iteration of our tool chain, a pull request affecting a single component always caused a rebuilt of the whole CPU to which the corresponding instance was allocated to. With growing number of developers contributing to the model, build times increased significantly, resulting in long waiting times and overall decrease of productivity. Only considering changed model entities in the check of a pull request eliminated these problems.

B. Incorporating legacy systems

Some data required to generate a complete system description is located into other data sources. For example, at BMW, the ethernet-based vehicle network description is managed in a separate database system, which is involved in a variety of processes and projects. The tool does not allow agile workflows, therefore we decided to import its data into our model on a regular basis. Whenever a release takes place for the ethernet-based vehicle network, we convert the artifacts into a *Franca+* model description and add them to our repository. With this approach, we decouple the code generation as described in V-A from the legacy system, so that its contents can be changed with our workflow. We did

not implement a round trip within our tools, though. In this example, changes introduced in the *Franca+* model have to be aligned with the other system by following their change processes on a regular basis.

The same approach applies for diagnosis description for *Unified Diagnostic Services* (UDS, ISO 14229). The origin of the data is a database system with a separate change process. We extended our tool chain to read an export of the diagnosis database on a regular basis and to convert it into the *Franca+* model. This was necessary because at many interfaces diagnostic data and requests have to be handled.

C. Tracing to other tools

Since graphical representations may help to understand the software architecture better and allow advanced documentation, we introduced another tool for the generation of *Unified Modeling Language (UML)* elements from our *Franca+* model.

We generate UML elements (data-types, components and interfaces) from the *Franca+* models and import them into an UML tool, in which other aspects of the software and system architecture are described. By this, we enable seamless tracing and referencing from other system aspects with are not defined in *Franca+*, towards the entities defined in *Franca+*. At the same time, this enables tracing from our Requirements Engineering tool to the entities defined in *Franca+*.

As the exchange format with these graphical UML tools we use *Eclipse UML2*, which is part of the *Eclipse Model Development Tools (MDT)* [15] framework. The *Franca+* metamodel can be mapped straight-forward to UML2, because most elements are defined in both representations.

D. Tool qualification

As we use the introduced architecture modeling approach for the development of mixed safety critical systems, an additional factor to consider is the tool qualification according to functional safety standards, particularly the ISO 26262 [16] for the automotive domain. The necessary actions, like determination of the tool confidence level (TCL) and software tool qualification, are currently ongoing.

In this section, we described the tool chain used to embed the *Franca+* model into the Continuous Integration (CI) infrastructure, to enable agile workflows and change processes. We described the translation from the model all the way to the binary running on the hardware. Additionally, we introduced tools used to incorporate legacy systems and generate UML model elements.

In the next section, we will introduce a concrete example, which illustrates our architecture development approach in daily practice.

VI. Development example

In the previous sections, we described modeling with *Franca+* and how agile teams and workflows can benefit from this approach. In this section, we take a dedicated piece of software which is part of the Software-Platform for Automated Driving and explain how it is developed with the tools and processes described.

A. Regulatory software identification numbers

To comply with statutory specifications and type approval regulations, the software platform shall provide regulatory software identification numbers (short: RxSWINs) [17] [18]. These numbers are used to identify the software version used during approval of the product and to establish a process to handle software updates and their impact on existing type-approved systems or functions. To comply with these requirements, a software shall be developed as part of the Software-Platform for Automated Driving, which handles RxSWINs and the relevant communication needs.

For the sake of simplicity in this illustration, our exemplary model below focuses on a single requirement derived out of the type approval regulations: *The software shall offer an interface to query the RxSWIN of the whole platform.*

B. Architecture Model of RxSwinApp

To fulfill the requirements derived for the software identification number feature of the platform, a service component called *RxSwinApp* is modeled, which offers a single interface containing a method. If the method is called, the RxSWIN of the platform is returned.

Thus, the *Franca+* model for this software contains a service component definition, including a provided port which offers interface RxSwinData. The interface contains one method definition with one out parameter. If the method is called by the client, the out parameter contains the RxSWIN. Listing 1 shows the model in *Franca+* language.

```
interface RxSwinData
{
  method getRxSwin
  {
    out {
      UInt32 rxswin
    }
  }
}

service component RxSwinApp {
  provides RxSwinData as RxSwinDataPPort
}
```

Listing 1. Definition of *interface* and *service component* in *Franca+*

To deploy *RxSwinApp* on a CPU and the interface *RxSwinData* towards a communication protocol, e.g. SOME/IP, additional parts of the model define the deployment of the entities defined in listing 1. For example the deployment on a CPU may include additional parameters which are required to configure the Adaptive AUTOSAR middleware for this application, e.g. regarding startup parameters or dependencies towards other applications.

C. Tool chain and software development for RxSwinApp

The *Franca+* model of *RxSwinApp* is stored in the overall project git repository. We use Bazel [19] as build system and to define a set of rules to compile the model into ARXML and C++ as described in section V. The corresponding C++ source code, which implements method *getRxSwin*, is stored in the same git repository, as well as all unit and component tests to ensure quality of the software.

Due to co-location of architecture model and source code and embedding everything into the Bazel build system and continuous integration, the agile workflows as described in section II-B can now be pursued whenever requirements are changing. Imagine for example a new requirement that requests returning the RxSWIN as character string, to be compatible with legacy testing environment. A feature team breaking down the new requirement may come up with the solution that introducing a second method returning a byte array instead of UInt32 is the most feasible solution for this use case. Due to model and code located in the same git repository and following the same change process, they can adapt the *Franca+* model, extend C++ implementation and add corresponding tests, all in a single pull request to be validated by the continuous integration system.

In this section, we have seen how Franca+ based architecture modeling is pursued on a real-life example within our project. Describing the software architecture with *Franca+* enables feature teams to change the model in the same way as the corresponding source code, which is a key prerequisite to apply agile processes, workflows and tools.

VII. Conclusion and Future Work

In this paper, we presented the requirements and a solution for a software architecture description approach in an agile development project. The solution included a *Treat Architecture like Code* philosophy and our architecture tool chain, based on an extended version of the publicly available *Franca* modeling framework.

Our *Treat Architecture like Code* approach facilitates feature teams to perform their end-to-end responsibility, from writing requirements, architecture descriptions, code, tests and eventually bring the changes into the master of the repository, such that the changes can be delivered. The approach enables a lean and efficient way to model modular independent parts of the architecture, by independent feature teams.

As success story, we can see already in the repository that our *Franca+* based modeling approach has been directly adopted in the development. More than 200 developers from 4 companies contributed to the *Franca+* models in quite a short time after releasing this approach. This is already a big improvement compared to traditional centralized UML based architecture modeling tools, where only some experts can edit the models (due to impediments in knowledge, rights or tool-licenses), and a simple change request sometimes takes weeks to introduce.

References

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 33–42.

[2] S. Fürst, "Challenges in the design of automotive software," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 256–258.

[3] W. Vieweg, *Management in Komplexität und Unsicherheit - Für agile Manager*. Berlin Heidelberg New York: Springer-Verlag, 2014.

[4] K. Ribel and C. Larman, "LeSS Huge at BMW Group," https://www.scrumalliance.org/ScrumRedesignDEVSite/media/pdfs/Konstantin-Ribel-181008_-_LeSS_Adoption_at_BMW_Group.pdf, October 2018.

[5] S. Fürst, "Autosar the next generation–the Adaptive Platform," in *3rd Workshop on Critical Automotive applications - Robustness and Safety (CARS@EDCC)*, 2015.

[6] L. Völker, "Some/ip-die middleware für ethernetbasierte kommunikation," *Hanser automotive networks*, 2013.

[7] M. Bregenzer, "LeSS adoption at a bavarian car manufacturer," https://less.works/case-studies/bmw-group.html, 2018, [Online; accessed 04-October-2019].

[8] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

[9] E. Woods, "Aligning architecture work with agile teams," *IEEE Software*, vol. 32, no. 5, pp. 24–26, Sep. 2015.

[10] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, and W. Hopfensitz, "On service-orientation for automotive software," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 193–202.

[11] Object Management Group (OMG), "Data-distribution-service," http://www.omg.org/spec/DDS.

[12] Genivi, "Franca framework," https://github.com/franca/franca, 2019, [Online; accessed 04-October-2019].

[13] E. Foundation, "Eclipse Xtext," https://www.eclipse.org/Xtext/, 2019, [Online; accessed 03-November-2019].

[14] itemis AG, "Franca User Guide," https://info.itemis.com/en/franca-user-guide, 2018, [Online; accessed 04-October-2019].

[15] E. Foundation, "Eclipse Model Development Tools," https://www.eclipse.org/modeling/mdt/?project=uml2/, 2019, [Online; accessed 15-November-2019].

[16] International Organization for Standardization (ISO), "ISO 26262-8 – Road vehicles – Functional safety, Part 8: Supporting processes," Technical Committee 22 (ISO/TC 22), Tech. Rep., 2011.

[17] "Regulation No 13-H of the Economic Commission for Europe of the United Nations (UN/ECE) - Uniform provisions concerning the approval of passenger cars with regard to braking," https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:42015X1222(01), 2015, [Online; accessed 04-October-2019].

[18] "Regulation No 79 of the Economic Commission for Europe of the United Nations (UN/ECE) - Uniform provisions concerning the approval of vehicles with regard to steering equipment," https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:42008X0527(01), 2008, [Online; accessed 04-October-2019].

[19] Google, "Bazel - a fast, scalable, multi-language and extensible build system," https://bazel.build/, 2019, [Online; accessed 04-October-2019].