Towards a Domain-Specific Language for the Virtual Validation of Cloud-native Mobility Services

1st Philipp Heisig
IDiAL Institute
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
philipp.heisig@fh-dortmund.de

2nd Christoph Flick
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
christoph.flick001@stud.fh-dortmund.de

***Abstract*—Future vehicles can be considered as "IoT devices on wheels" as they exhibit high-performance computation resources, various sensing devices, and a data-driven software architecture. While the availability of automotive big data provides the basis for innovative and disruptive mobility services, processing vehicle data within the cloud poses also several challenges. A major challenge in this context is the validation of cloud-native mobility services regarding their proper functionality and the fulfillment of non-functional requirements. Due to the cost-efficient nature of simulations, traffic simulation in combination with network simulation is more and more used for a virtual proof-of-concept of mobility services and their software architectures. Nevertheless, the creation of adequate simulation environments specific to connected vehicle scenarios is time-consuming and requires explicit domain knowledge. In this paper, we present a prototypical domain-specific language tailored to the formal description of connected vehicle scenarios and the according generation of simulation environments. Therefore, we make use of the traffic simulator Eclipse SUMO as well as the co-simulation environment Eclipse MOSAIC and demonstrate the usage of our DSL via the use case of a restricted traffic zone. Although the DSL so far only support the setup of minimal traffic scenarios, it already helps to abstract complexity and ease the set-up of simulation environments for connected vehicle scenarios.**

***Index Terms*—Connected Vehicles, Simulation, Cloud Computing, IoT, Domain-specific Language, Software Modeling**

I. Introduction

Technological advances, digitization, and area-wide mobile Internet have transformed vehicles into software-based high tech products with built-in connectivity and autonomous driving features. These vehicles are characterized by the massive amount of multi-modal data emitted by hundreds of various sensors [1] to provide context information about the vehicle itself and its environment, e. g. to detect road conditions, monitor tire pressure, or recognize driver's fatigue. Sharing these data within the Internet of Things (IoT) by means of connected vehicles facilitates vehicle data collection and processing at scale in multiple and simultaneously operating services running in the cloud. By utilizing cloud computing capabilities for data fusion, analysis, and processing, innovative and data-driven mobility services running in the cloud can be realized, spanning from road safety over smart, efficient, and green transportation to location-dependent services.

However, connected vehicles operate in a safety-critical and time-sensitive environment with changing conditions. Unreliable vehicle connectivity with changing data transfer rates must be expected and real-time processing of the resulting data may be necessary for certain features. Furthermore, cloud-native mobility services have to scale with the high number of vehicles on the road, while the architecture has to process also a variance in data stemming from different types of vehicles. Hence, scalable, resilient, and secured mobility services are required to achieve the vision of connected vehicles.

A major challenge here is to test and validate mobility services regarding their proper functionality and the fulfillment of non-functional requirements, in particular scalability and reliability. While occasionally applied test approaches in the automotive domain focused on the vehicle itself and the according validation of in-vehicle functionality, cloud-native mobility services operate upon networks of vehicles and infrastructure devices and thus require a massive amount of varying and vehicle-specific data that need to be fed into the services for testing. As traffic scenarios are manifold and complex, test drives based on a vehicle fleet or setting up a large number of hardware and vehicle nodes to generate vehicle-specific data are not feasible from an economic and operational perspective, whereas dummy data lack of semantics and neglect environmental conditions like changing connectivity.

A promising way to still establish a testing process for cloud-native mobility services is the usage of traffic simulations to generate context-specific automotive data that goes beyond rudimentary or fake data-sets. In combination with network simulators, connected vehicle scenarios can be virtually created for a proof-of-concept design and evaluation. Due to the cost-efficient nature of simulations, virtual testing can be carried out through the whole development process and feedback loops based on simulation data can be established to test and improve services again and again. Such feedback is especially valuable at early stages of a development process as it provides crucial insights on potential problems with the defined software architecture or used technology and prevents costly and complex late-lifecycle changes [2].

Nevertheless, simulators are usually specialized in reproducing certain aspects, but at least traffic and network simulators have to be interconnected to support all of the previously described validation aspects for connected vehicle scenarios. Co-simulation refers to the coupling and composition of multiple

simulators to simulate an overall system [3]. Setting up such co-simulation environments is, however, a complex and time-consuming task that requires a lot of domain knowledge and may prevent developers and software architects from focusing on the actual service implementation. As stated by Ersal et al. [4], more research is needed in usability and specification validation/testing. Especially small and medium-sized enterprises (SMEs), cities and municipalities, or service providers yet outside of the automotive domain, such as insurances, are potential candidates for realizing innovative and cross-domain mobility services, but lack expertise in automotive testing and validation approaches.

To ease the set-up of virtual testing environments and thus enable the development of cloud-native mobility services, we propose a model-based scenario description via a domain-specific language (DSL). Therefore, we define a first prototype of a DSL in this paper that can be used to generate basic traffic scenarios as foundation for a virtual testing environment. In addition, we evaluate the applicability of the DSL via the use case of a restricted traffic zone service. To run the simulation, we make use of the co-simulation framework Eclipse MOSAIC[1] along with the traffic simulator Eclipse SUMO[2] and Eclipse MOSAIC's Simple Network Simulator for simulating ad hoc communication.

The remainder of this paper is organized as follows: Section II introduces a general approach on how to enable virtual testing of cloud-native mobility services. Afterward, Section III proposes a first version of a DSL to formally describe traffic scenarios along with a generator for SUMO simulator configurations. Section IV then evaluate the applicability of the DSL via the use case of a restricted traffic zone service, while Section V discusses the results and potential drawbacks. Finally, Section VI concludes this work.

II. Virtual Testing Cloud-native Mobility Services

This section introduces a model-based approach to ease the set-up of co-simulation environments for connected vehicle scenarios. The overall goal is to provide sufficient simulation data for testing cloud-native mobility services by generating an adequate simulation environment via a model-based scenario description. As shown in Figure 1, the approach basically consists of four steps that are described in the following:

1) *Scenario Specification:* The first step is the definition of a connected vehicle scenario including its requirements, which is usually done by a domain expert in natural language. Based on this, software architects can start to define a first sketch of a software architecture and developers may implement basic functionality. Likely, the architecture will be based on a microservice architecture (MSA) to feature scalability and flexibility [5].
2) *Modeling and Configuration:* Within the second step, the informal *Scenario Specification* will be formalized via a DSL that is designed for describing connected vehicle scenarios, e. g. how many and what types of vehicles should be simulated. In addition, the DSL can be extended to capture non-functional requirements towards cloud-native mobility services like scalability. The resulting *Scenario Model*, i. e. a model that conforms to the DSL's metamodel, acts as input for a set of *Config Generators*, which automatically generates configurations for each simulator used in the scenario. Depending on the simulation tool and in which way it can be configured, model-to-model or model-to-text transformations can be applied for the generation process.
3) *Co-simulation:* While the configurations allow to set up each simulator independently, they still need to be integrated into a co-simulation environment including a component that is responsible for orchestrating and controlling the simulation flow of each simulator to enable interoperability among the simulators. For this step, existing open-source co-simulation frameworks like Eclipse MOSAIC or Eclipse OpenMCx[3] are potential candidates to be used. The simulation environment then generates large amount of semantically enriched vehicle data on different levels of detail. For replication of tests or to carry out tests at any time, the generated data will be persisted in a database.
4) *Feedback Loop:* In the last step, test data from the simulation will be contentiously fed into the mobility service to test both the service functionality and the software architecture behind it. Predefined metrics assess the architecture against the different non-functional requirements defined in the *Scenario Model*, such as response time or the number of vehicles that have been simultaneously served. The test results are generated by a *Report Generator*, which helps the developer to improve the service implementation and software architecture behind it. By repeating the test, either via new or based on the previous data sets, *Feedback Loops* can be established.

III. Domain-specific Language for Connected Vehicle Scenarios

While the previous section introduced the overall testing approach for connected vehicle scenarios, this section aims at realizing the second step of the approach (*Modeling and Configuration*) by proposing a first draft of a DSL for generating multi-modal traffic scenarios.

In general, DSLs are languages designed to describe and solve problems within a specific application domain. DSLs helps domain expert to read, understand, and even write code via formalized domain models that conform to the metamodel of the corresponding DSL. In combination with code generators, DSLs are powerful tools to abstract the underlying complexity of software systems and enhance development

[1] https://www.eclipse.org/mosaic/
[2] https://www.eclipse.org/sumo/
[3] https://projects.eclipse.org/projects/automotive.openmcx

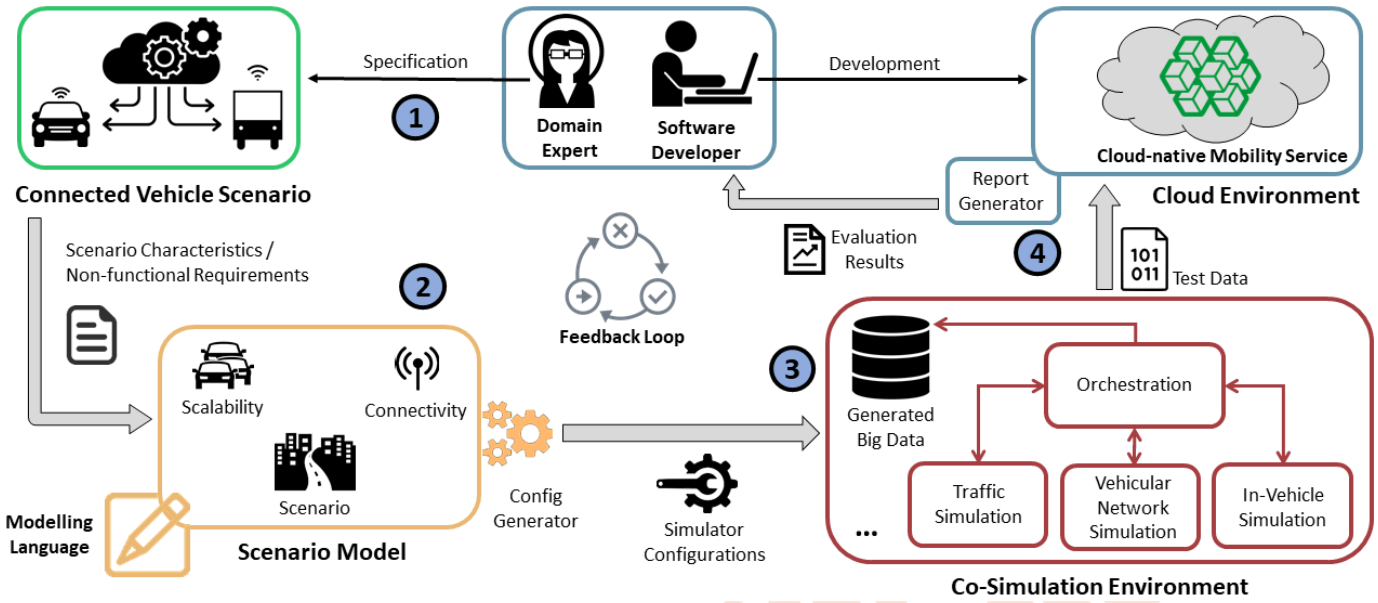

Fig. 1. Model-based testing approach for connected vehicle scenarios

efficiency. Thus, we propose the usage of a DSL to formally describe requirements for connected vehicle scenarios.

However, connected vehicle scenarios are complex and involve a lot of varying and uncertain factors, e. g. different vehicles with different configurations are interacting among each other and with further traffic participants and infrastructure such as cyclists, pedestrians, traffic lights. This leads to a tremendous number of possible scenarios that need to be tested [6]. Therefore, we focus in this paper on the definition of minimal traffic scenarios for a first proof of concept. The scope of the DSL is to provide a simple, straightforward way to setup a traffic simulator with scenarios and simulator settings.

For the first prototype, our DSL will integrate a code generator (*Config Generator*) for the open-source traffic simulation suite Eclipse SUMO, which is designed for microscopic simulations and supports, among other things, large road networks and the modeling of inter-modal traffic systems including vehicles, public transport, and pedestrians. In addition, Eclipse SUMO is well established in the research community and provides real-world scenarios. For the implementation of the DSL, Eclipse Xtext[4] is used as it is also open-source and provides quality-of-life features such as the editor support. Listing 1 shows an excerpt of the Xtext grammar for the DSL, while Table I depicts the different building block in detail.

The foundation of the DSL is built by one or more `configure` blocks which contain the configuration details for a specific simulator, e. g. `configure SUMO` could be used to start a configuration block for SUMO. For each `configure` block, and as such for each simulator, a separate configuration file will be created based on predefined *Config Generator*. Therefore, the generator traverses the model and converts the data from the DSL into simulator-specific configuration values. In the case of SUMO, for example, the file generation feature of Eclipse Xtext is used to create a *.sumocfg* file, which is the central configuration for a SUMO scenario. As the DSL is intended to rely on open standards and reuse existing tools as much as possible, the road network generator *netgenerate*[5] in combination with the *randomTrips.py* script is used to generate a road network as well as random traffic demand, respectively, in the form of XML files. In addition to the DSL's capabilities to configure the generation of network and traffic demand, it also allows to refer existing files alternatively, e. g. from real-world scenarios. Apart from the ability to describe network and traffic demand, the DSL offers more configuration options for the simulation in general, e. g. the user of the DSL can set the step length of the simulation to influences its granularity.

Although traffic simulation provides already a meaningful set of vehicle data for testing, coupling at least a traffic simulator with a network simulator within a co-simulation environment is necessary for simulating connected vehicle scenarios. Eclipse MOSAIC[6] is an open-source, multi-domain, co-simulation framework for connected and automated mobility that supports the integration and coupling of simulators from different domains. To run a basic traffic scenario within Eclipse MOSAIC, a SUMO network file and a SUMO configuration file are required, whereas the route file is created by the *MOSAIC SUMO Ambassador* during execution. While these files can be already generated with the SUMO *Config Generator* of our DSL, running SUMO within MOSAIC requires further files such as a *runtime.json*, a *mapping_config.json* for mapping simulated applications to the simulation units, and a scenario database, which contains

[4]https://www.eclipse.org/Xtext/

[5]https://sumo.dlr.de/docs/netgenerate.html

[6]https://www.eclipse.org/mosaic/

data about the roads, road connections, potential road restrictions, and all possible routes which vehicles may take in the simulation. To also automate the setup of these files and run SUMO scenarios within Eclipse MOSAIC, one can use the `MOSAIC` mode of our DSL, which generates the required files with default values. The mode also enables the simulation of existing SUMO traffic scenarios with MOSAIC via the `SumoScenarioAmbassador` instead of using a scenario database. Basically, the `SumoScenarioAmbassador` implements a TraCI[7] client that sends all relevant commands to the running SUMO instance.

To further improve the portability of the simulation, a docker mode was introduced to the language, which can be enabled by putting `mode Docker` at the top of a file. When the docker mode is active, not only the SUMO files are generated but also a Dockerfile which contains the commands to create a fully-functional image including the desired simulators and the setup for those generators. Besides, a *README.md* gets generated with instructions and commands to run the dockerized simulation suite.

Listing 1
EXCERPT OF THE DSL'S XTEXT GRAMMAR

```
Domainmodel:
    ('mode' mode=Mode)?
   config+=Config+;

Config:
   'configure' name=Simulator '{'
      (input=Input &
      output=Output? &
      time=Time? &
      routing=Routing?)
   '}';

Input:
   'input' '{'
      input=(FileInput | GeneratorInput)
   '}';

GeneratorInput:
   'generate' type=GeneratorType 'size' size=INT;
   ('random-seed' randomSeed=INT)?;

FileInput:
   {FileInput}
   (('netFile' netFile=STRING) &
   ('routeFiles' routeFiles=List)? &
   ('additionalFiles' additionalFiles=List)?);

Output:
   {Output} 'output' '{'
      ((humanReadable?='humanReadable')? &
      ('statisticFile' statisticFile=STRING)? &
      ('summaryFile' summaryFile=STRING)? &
      ('tripinfoFile' tripinfoFile=STRING)?)
   '}';

Time:
   {Time} 'time' '{'
      (('start_at' start=INT 'seconds')? &
      ('end_at' end=INT 'seconds')? &
      ('steplength' steplength=DOUBLE 'seconds')?)
   '}';

Routing:
   {Routing} 'routing' '{'
      (('algorithm' algorithm=Alogrithm)?)
   '}';

List:
   list+=STRING (',' list+=STRING)*;

enum Mode:
   Simple | Docker | Docker_TraCI | MOSAIC | MOSAIC_Docker;

enum GeneratorType:
   Grid | Spider | Random;

enum Alogrithm:
   dijkstra | astar | CH | CHWrapper;

enum Simulator:
   SUMO;
```

Table I
DESCRIPTION OF THE DIFFERENT DSL BUILDING BLOCKS

Building Block	Description
`Mode`	The mode for the code generator. Possible values are `Simple`, `Docker`, `Docker_TraCI`, `MOSAIC`, and `MOSAIC_Docker`. When the code generator is executed in `Simple` mode, the simulator configuration is generated without any extras. MOSAIC tells the code generator to generate additional files to run SUMO scenarios in the co-simulation environment MOSAIC. The `Docker`, `MOSAIC_Docker`, and `Docker_TraCI` modes additionally generate a Dockerfile to run the simulation. The latter will also generate a configuration which runs the SUMO simulator with a port open for TraCI.
`Config`	Includes the configuration properties for a specific simulator. It has to be used exactly once for each simulator to be configured.
`Input`	Provides the input data for the defined simulators in `Config`. The input can be described either via a `GeneratorInput` or a `FileInput`.
`Generator Input`	When `GeneratorInput` is provided, the code generator also generates road networks and traffic demand in addition to the configuration. To achieve that, a `GeneratorType` and a size for the network need to be provided. Additionally, a seed can be set for the random generator so the result can be regenerated.
`Generator Type`	The type of road network that needs to be generated. The options here are (i) `Grid` to generate a network with a grid layout; (ii) `Spider` to generate a network in form of a spider's web; or (iii) `Random` to generate a random network without a predefined layout.
`FileInput`	Tells the code generator to use the configuration files provided in this block. The files that can be set here are the (i) `netFile` that contains the description of the network; (ii) one or more optionally `routeFiles` which describe the routes simulations entities may take; and (iii) `additionalFiles` as placeholder for various purposes.
`Output`	Contains properties to define the output of the simulation. For example, if and where statistics or a summary file should be produced.
`Time`	The `Time` block configures time-based properties. In particular, the start and end time of the simulation scenario, which control the overall running time, can be set. Furthermore, the duration of each simulation step can be influenced.
`Routing`	The `Routing` block contains properties to influence the routing behavior of the simulation. The main setting here is the routing algorithm to be used, which can be either dijkstra, astar, CH, or CHWrapper [7].

[7]https://sumo.dlr.de/docs/TraCI.html

IV. Use Case: Restricted Traffic Zone

This section introduces the use case of a restricted traffic zone to demonstrate and evaluate the DSL prototype defined in Section III. In this use case, unauthorized vehicles are prevented from entering a predefined area for safety reasons, e. g. when a festival is taking place. Therefore, a roadside unit is placed near or inside the restricted zone that communicates with all vehicles close by via ad hoc close-range communication. The roadside unit request the vehicle type and ID from each vehicle, calculates the distance of the vehicle to the restricted traffic zone, and send all data to a cloud-native service deployed by the city's traffic department. The service receives the data and checks if the vehicle is allowed to enter the zone, e. g. vehicles like an ambulances should be allowed to enter the zone. If the vehicle is not authorized and violates the rules by approaching too close to the restricted traffic zone, the service broadcasts a stop signal via the roadside unit to the vehicle, which has an in-vehicle application deployed that can process the stop signal and execute it.

This use case involves different types of vehicles and devices with software running on embedded devices as well as within the cloud. Developing such a service requires a through investigation and developers implementing the service have to assess the functionality and software architecture behind already at early stages of the development process. To save time, money, and ensure high-quality services, a first proof of concept via a simulated environment can give valuable feedback about the service quality and potential technical problems. Based on the DSL (see Listing 1), a rudimentary co-simulation environment for the use case can be defined as shown in Listing 2, which basically creates a SUMO traffic scenario. Among the traffic simulation via Eclipse SUMO, also the ad hoc close-range communication between vehicles and roadside unit as well as the in-vehicle application have to be simulated. Thus, we make use of the co-simulation framework Eclipse MOSAIC and accordingly set the `MOSAIC` mode of our DSL to further integrate the according simulators.

Listing 2
Example *Scenario Model* for the restricted traffic zone use case based on the DSL

```
mode MOSAIC

configure SUMO {
   input {
      netFile "highway.net.xml"
      routeFiles "highway.rou.xml"
   }

   time {
      start_at 0 seconds
      end_at 1000 seconds
   }
}
```

For the simulation of the in-vehicle application, a predefined Java class implementing the `VehicleOperatingSystem` interface is added in the generated *mapping_config.json* file. The application simply reduces the vehicle speed in case it receives a stop signal. Within the *mapping_config.json* file, also the location of the roadside unit can be set via latitude and longitude values. The communication between the roadside unit and vehicles is based on IEEE 802.11p and can be simulated by using the Simple Network Simulator, which is already integrated in Eclipse MOSAIC and can be configured within the *sns_config.json* file, e. g. the communication range can be altered by setting the maximum allowed number of hops. For the implementation of the roadside unit application, Eclipse MOSAIC integrates another interface, called `RoadSideUnitOperatingSystem`, that provides functionality specific to roadside units, e. g. vehicle routing. The application creates a loop in which it broadcasts a message every few milliseconds to all vehicles inside the prior defined geographical area using the `GeoArea` class.

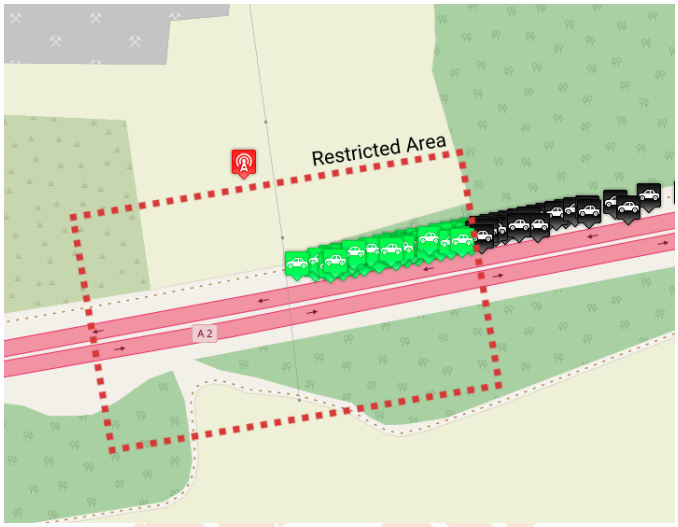

Fig. 2. Visualization of the simulation for the restricted traffic zone use case

Figure 2 shows the running simulation of the defined use case, which is the third step of the process in Section II. During the simulation, vehicles were spawned on a highway and then attempted to drive through the restricted area. In the figure, simulation units sending messages are colored in red while units receiving messages are colored green, i. e. vehicles inside the area are receiving messages, and the roadside unit is sending messages. Due to different vehicle speeds, the braking distance of each vehicles differs accordingly.

V. Discussion

The use case in the previous section demonstrated that our DSL prototype allows a highly expressive description of basic traffic scenario configurations by means of road networks as well as traffic demand. By using the code generation facilities of Eclipse Xtext, SUMO simulation scenarios can be generated and also integrated in the co-simulation environment Eclipse MOSAIC. In addition, we support dockerized images for greater portability. Generally, using a DSL improves productivity and reduces effort when creating traffic scenarios.

Nevertheless, there are several challenges and drawbacks we observed when creating the DSL. One central issue when creating a DSL for the description of traffic scenarios is the

complexity of the domain, especially when the goal of the language is the generation of concrete simulation scenarios. The complexity of the DSL and its functionality have to be carefully balanced so that it is powerful enough to describe traffic scenarios in a sufficient manner, but still be easily graspable by domain experts and parsable by a code generator. If the DSL becomes too complex, it would lose most of its intended purpose. Essentially, the only remaining advantage would be the tool support that an DSL provides, such as auto completion and syntax highlighting.

For a first prototype, our DSL was narrowed down to the usage with the traffic simulator Eclipse SUMO. This focus led to a bias during the design of the DSL, resulting in the DSL's structure to be similar to the structure of configuration files of SUMO scenarios. While this DSL was a first prototype for a general proof-of-concept if formal description of traffic scnearios in the context of connected vehicles are applicable, future versions of the DSL would need certain extensions to better describe general-purpose traffic scenarios.

There are also some drawbacks when running a SUMO scenario within Eclipse MOSAIC via the `SumoScenarioAmbassador` instead of having a proper scenario database file: i) In-vehicle applications can only be mapped to vehicles types, but not individual vehicles; ii) all simulated vehicles are spawned by SUMO which prevents independent vehicle spawners; iii) simulated application are not able to make use of the navigation module to influence vehicle's routing in the applications; and iv) it is not possible to map applications to traffic lights. Another concern when using the application simulator in Eclipse MOSAIC to test a connected vehicle application is that it need to be wrapped with a MOSAIC application to be testable, which only work with Java applications currently.

VI. Conclusion

In this paper, we presented a first prototype for a DSL that supports the formal description of connected vehicle scenarios and the automatic derivation of a simulation environment for the traffic simulator Eclipse SUMO as well as the co-simulation environment Eclipse MOSAIC via generators. Despite that the DSL so far only support the setup of minimal traffic scenarios, we have shown with the use case of a restricted traffic zone that it is already possible to test more sophisticated connected vehicles applications that require the coupling of different simulators. By providing Docker support, the setup of simulation environments can be significant simplified. Nevertheless, there are still several drawbacks as discussed in Section V. Generally, a more generic, simulator-independent approach is necessary that also integrates domain-independent testing aspects like data privacy. Therefore, a stronger focus should be placed on the integration of available open-source formats as the DSL foundation, such as ASAM OpenCRG, OpenDRIVE, and OpenSCENARIO

For the future we are planing to redesign our DSL regarding more flexibility and provide a ready to use web-based user interface without the hassle of installing and configuring the DSL locally. Furthermore, more sophisticated end-to-end connected vehicle scenarios that integrate additional data sources, e. g. from traffic infrastructure or the smart city, are required to extend our DSL with new building blocks. This also includes the support of additional simulators and automotive standards such as the Vehicle Signal Specification[8]. In addition to that, we want to define metrics to asses the architecture against non-functional requirements as described in step four in Section II. Therefore, metrics specifically developed for MSAs can be integrated to identify, for example, microservices anti pattern [8] such as API versioning or hard-coded endpoints. Another example for a MSA-specific metric would be to measure the cyclic dependency, i. e. the amount of inter-service communication.

References

[1] Leandro D'Orazio, Filippo Visintainer, and Marco Darin. Sensor networks on the car: State of the art and future challenges. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[2] Byron J Williams and Jeffrey C Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

[3] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *arXiv preprint arXiv:1702.00686*, 2017.

[4] Tulga Ersal, Ilya Kolmanovsky, Neda Masoud, Necmiye Ozay, Jeffrey Scruggs, Ram Vasudevan, and Gábor Orosz. Connected and automated road vehicles: state of the art and future challenges. *Vehicle system dynamics*, 58(5):672–704, 2020.

[5] Tobias Schneider and A Wolfsmantel. Achieving cloud scalability with microservices and devops in the connected car domain. In *Software Engineering (Workshops)*, pages 138–141, 2016.

[6] Sven Hallerbach. *Simulation-based testing of cooperative and automated vehicles*. PhD thesis, Universität Oldenburg, 2020.

[7] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*, pages 2575–2582. IEEE, November 2018.

[8] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Microservices anti-patterns: A taxonomy. In *Microservices*, pages 111–128. Springer, 2020.

[8] https://genivi.github.io/vehicle_signal_specification/