Test Automation Architecture for Automotive Online-Services

Robin Steller
Beuth University of Applied Science, Berlin, Germany
Email: robin-steller@web.de

Marek Stess
Volkswagen Commercial Vehicles, Hanover, Germany
Email: marek.stess@volkswagen.de

***Abstract*—The car industry undergoes a big change, triggered by modern agile methods for software development. New software features can be rapidly developed and instantly deployed. In contrast to that stands the long development cycle of the vehicle and its hardware. That hinders the development speed due to hardware related dependencies. Vehicle hardware requires extensive security measurements and procedures. Software embedded directly on the hardware holds many dependencies, varying for different car types within the same manufacturer. In the typical software development an Operating System (OS) diminishes these hardware dependencies and in the means of web development even OS dependencies can be resolved, enabling cross platform development. Therefore the following paper focuses on the development of a test automation architecture. It depicts how our architecture can be built to combine long term vehicle development with rapid agile software development and integrate them together. We call this DeepTesting, the complete test coverage of the software system with its services and applications. Additionally the DeepTesting architecture will consume real world test data to test software under nearly real world condition.**

***Index Terms*—automation, testing, architecture, continuous integration**

I. Introduction

In the last years mobile services have an important factor in the automotive industry. The desires for autonomous driving, passenger safety and user entertainment have pushed the development of the connected car and thus the need for online services, communicating and exchanging data with the car over the air [1], [2].

Volkswagen Commercial Vehicles currently develops a mobile fleet management system called ConnectedVan [3]. It enables users and companies to easily monitor their vehicles. The user can record driver's logbooks, fuel logs, statistical reports and more.

The basic architecture behind the fleet management system ConnectedVan is depicted in Fig. 1. In general we differ between two ways of data communication. Data can be collected via the Online Connectivity Unit (OCU).

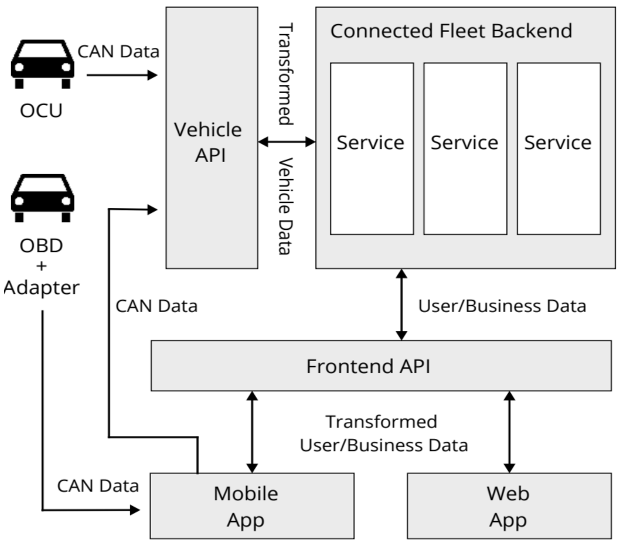

Figure 1. Basic Architecture of ConnectedVan

In older car models the vehicle data is transmitted via the Onboard Diagnostics (OBD) and an OBD adapter. The OCU is equipped with a SIM card to allow communication Over-the-air. Starting in 2018 new Volkswagen cars feature the OCU by default. The OBD adapter is connected to the OBD and sends data to the ConnectedVan application running on a bluetooth paired device. Both, OBD and OCU, read out the signals of the CAN bus (Controller Area Network) in the car. The CAN serves as the main communication network for signals within a vehicle [4].

All data collected from the vehicle is sent to the Vehicle API (VAPI) for data transformation. The reason for data transformation originates in the large variety of car models, each providing different forms of object models for their data, especially in older car generations. To ensure object conformity for all backend services, the VAPI converts all data to a standard object model.

After transformation the data is forwarded to the Connected Fleet Backend (CFB) which serves as the central business logic. The CFB holds all relevant

Manuscript received August 21, 2018; revised May 10, 2019.

backend services like vehicle-service, driver-service, logbook-service and many more.

Due to multiple different devices and formats like smart- phone, tablets and desktops, data will be further transformed by the Frontend API (FAPI). That way all business and user data is provisioned in a standardized way to all applications. Current applications are the Android, iOS and web app.

In this context following aspects become a problem in terms of testing.

- Real world condition test are either not feasible or require a non profitable effort. Testing the whole system close to the reality would require thousands of cars constantly performing test drives to supply valid testing data. To the respect of real time data transmission and the goal of constantly available data, every kind of test drive (fast, slow, crash, extreme weather conditions, etc) would have to be performed at all times.
- Providing test coverage to the whole system is not trivial. The architecture holds hard dependencies on actual hardware (OBD, OBD adapter, OCU, smartphone, vehicle, CAN, CAN adapters), backend (APIs, micro services) and frontend (desktop & smartphone client). Common test automation architectures, explained later in the paper, cover isolated components of such architecture or can't resolve hardware dependencies.
- As a result of the previous point, embedding the infrastructure into a continuous integration cycle proves as challenging due to the variety of dependencies. The goal is not only to test each part of the architecture in isolation, but to enable a deep test through the whole system. That testing needs to be automated and integrated into the CI process of daily software development.

By overcoming these problems and integrating a DeepTesting architecture the speed of development and the software quality will increase gradually. Big efforts of manual testing can continuously be put into the actual software development.

A. *What Is Test Automation?*

Test automation enables the developer to execute tests on intervals or events, without having to perform any manual task. That can be a routine running on a daily base or an event triggering the test process e.g. new commit [5]. As a definition of test automation we can follow the statement that “test automation is the task of creating a mechanically interpretable representation of a manual test case” [6]. Automated tests should record results that indicate whether the test passes or fails [5].

Different kinds of test types can be automated. They will be mentioned in the further subsection. The industry aims to automate as much of their test activity as possible to increase the test coverage and quality, but also to reduce costs for human resource, bug fixing, etc. The expected increase of the test automation market will be 23,01% by 2022 [7]. Software testing itself makes up 50%-60% of the total cost of software development [8], [9].

B. *Types of Testing*

Automation can be applied to various types and dimensions of testing routines in software development. Following types will be supported by the final DeepTesting architecture.

- **Unit Test**
  Unit tests check single functionalities and methods of a software module, verifying its correctness [10]. Test data is usually provided e.g. mocked by the developer [11]. Possible test designs can be both, black-box and white-box testing [12], [13].
- **Integration Test**
  Integration tests ensure the correct functionality of multiple components working together [14]. That could be a test, confirming the API's correct communication with the database. Common integration test designs are top-down and bottom-up [15].
- **Regression Test**
  Goal of the regression test is the detection of any (re)occurring errors, caused by newly added code or functionalities [8]. As a result, the whole application must be tested thoroughly. Regression test is the most time/effort consuming type of testing and therefore predestined for automation [16].

C. *Main Contribution*

Because of the complexity of vehicles, the various types of data that the vehicle transmits and the complex system landscape of mobile online services it's important to create an architecture that empowers end-to-end test automation. The DeepTesting architecture will enable complete testability across multiple APIs, frontends and services at once. That way multiple software components can be integrated more easily, due to system wide test coverage. It also enables testing of software without any dependencies on the actual extensive car development cycle and thus increases quality and velocity of development.

The paper is structured into following Sections. Section II analyzes the current state of the art in test automation architectures. Section III outlines different approaches for test data generation. Section IV describes the DeepTesting architecture resolving the problems displayed in Section I. The Section V displays the conduct of the testing process and management. Section VI examines the impact of the DeepTesting architecture on the infrastructure and development. Section VII concludes this paper and proposes future improvements.

II. State of the Art

This section depicts the currently utilized types of test automation and its testing domain. To the best knowledge of the author no publication of test automation infrastructures or architectures for online services in the automotive industry could be retrieved for this paper. Publications available regarding automation in automotive industry exclusively considered the software or hardware of the vehicle itself.

A. Netflix

Netflix developed an architecture for automated testing on various types of devices. Background is the huge variety of gadgets Netflix is supporting with software. Netflix runs its SDK on millions of devices, including gaming consoles, TVs/STBs, tablets, smartphones, laptops and computers [17]. The critical concept behind Netflix's setup is to reduce complexity of creating/running test cases to ensure high agility [17].

As displayed in Fig. 2 the architecture contains three major components: automation services, test suites and test devices. The automation services are a set of external backend services supporting the device handling, execution of tests and providing external features for testing. The connected test devices are connected to a Test Portability Layer (TPL) to unify device specific operations on an abstract layer. As a result different devices can be engaged via the same interface. The test runner supports the process of calling out to specific devices and services needed for a test run. The test suite only specifies the type of device and services. The runner manages the communication.

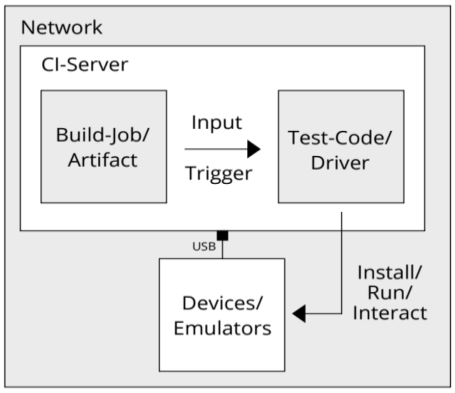

Figure 2. Test architecture for GUI by Thalia [19]

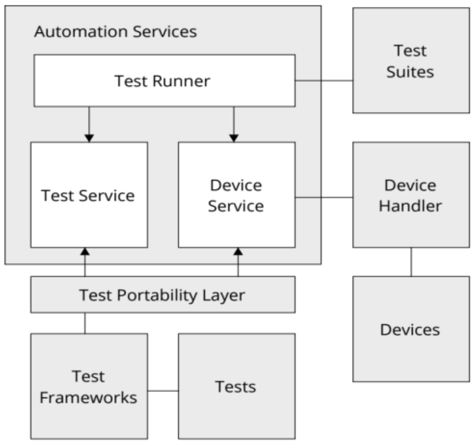

Figure 3. Test architecture for devices by Netflix [17]

B. Thalia Holding

Thalia, Germany largest book shop franchise [18], created a test automation architecture for automating regression tests against the GUI of their android and iOS applications. The infrastructure consists of a CI server, where the build of the applications and the test routines are executed, see Fig. 3. The build job serves as a trigger for the testing routine. As soon as a build of the application succeeds, the artifact of the build job, e.g. Android Package Application (APK) or iOS App Store Package (IPA), is forwarded to the test job. After a finished test run, the results of all tests are collected and presented on the CI server [19]. The GUI tests are designed as black-box tests and run as regression-tests [19]. The tester only examines and tests the surface without regarding the underlying internal structure of code [20].

C. Summary

All of the above listed examples for test architectures lack the capability of performing tests throughout the whole system. They focus on single parts of the overall system and test them in isolation, providing mock data or similar. The setups also neglect one important dependency and that is the dynamic data input which originates from the vehicle in our case. Also, the data supplied in the architectures is static and doesn't need to be generated or supplied in real time when executing test cases.

III. DATA

A. Test Data Generation & Communication

Test data is generated by creating a dump of the CAN bus data transmitted by the car. Data can be extracted via the logging interface and the CAN bus by multiple hardware devices. Hardware developed and currently in use at Volkswagen are CarGate, Car Telematics (CaTe) and a prototype called Car on Demand Interims Solution (CDIS) [21]. These devices are connected to the logging interface of the car and receive all data sent through the CAN bus. The devices differ in performance and transmission rate. CarGate features one low and one high speed CAN-Port whereas the CaTe consists of one low and four high speed CAN-Ports [21]. CDIS is the most advanced hardware containing two low and four high speed CAN-Ports.

The data received through the mentioned devices can be accessed by connecting a Linux computer via TCP/IP i.e. LAN [21]. The Linux cantools **candump** and **cangen** are then used to extract and bundle the vehicle data into a logfile, the CAN dump [22].

Another option of test data generation is a TestRack in combination with CANoe distributed by Vector. The TestRack consists of multiple vehicle sensor and control units employed in real cars. TestRack can simulate the behavior, on signal level, of a car. The software CANoe can read, manipulate and send data through the CAN of the TestRack. Through user inputs such as varying speed, car model, directions or fuel type, different test drives can be performed and recorded. This however, collides with the idea of providing data close to the reality, but is applicable for early development and testing stages of the architecture.

B. VirtualCarGate

VirtualCarGate (VCG) is a Debian virtual machine which simulates an operating car. As illustrated in Fig. 4 VirtualCarGate consists of various cantools, an EXLAP service and the generated logfiles. VCG utilizes the cantool **canplayer** to simulate the needed CAN interface [22], [23]. The **canplayer** then replays the complete content of the logfile until the logfile exceeds or the process is stopped.

VirtualCarGate features the Extensible Lightweight Asynchronous Protocol (EXLAP) Service for communication with other EXLAP-clients. EXLAP is a protocol for client/server communication. EXLAP makes use of a simplified XML scheme. VCG transmits its data that is induced by the logfile and replayed by the canplayer, via an EXLAP service. Other EXLAP clients can subscribe to that service. The data will only be published if the value of the subscribed property has changed since its last publication [24].

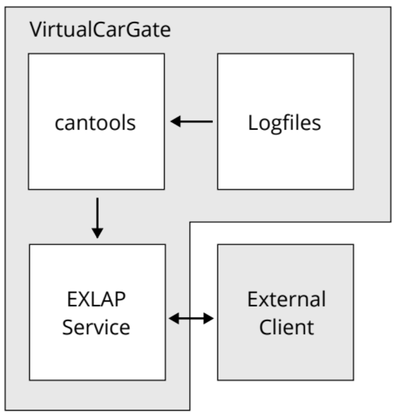

Figure 4. Test data simulation and distribution

IV. Deep Testing Architecture

This section outlines the DeepTesting architecture, depicted in Fig. 5, with its single components and employed technologies. The architecture consists of the VirtualCarGate for data supply, the Playback Server to trigger the test runs and the CI Server which stores and runs the test scripts. The architecture envelopes the ConnectedVan system mentioned in Section I and seen on Fig. 1.

A. Playback Server

The Playback Server will serve as the data and test management center. It should enable the user (developer, QA engineer) to select a specific vehicle or desired test drive and a set of tests that should be executed. The playback server will then start a VirtualCarGate instance to run the simulation process and channels the data to the specific test set. If the data stream is provided, the PlayBack server will then trigger the test job to run on the CI server. If the test job is finished the playback server will shut down the VirtualCarGate instance. The set of tests selected on the playback server defines which parts of the overall system will be needed for testing. That could be a set running against a single specific service or API while the GUI test job verifies the correctness of that supplied data on the frontend.

The Playback Server also gives feedback about the test result to the user. The feedback consists of simple data whether the test fails or passes. Additionally the occurred error is published and its location within the system.

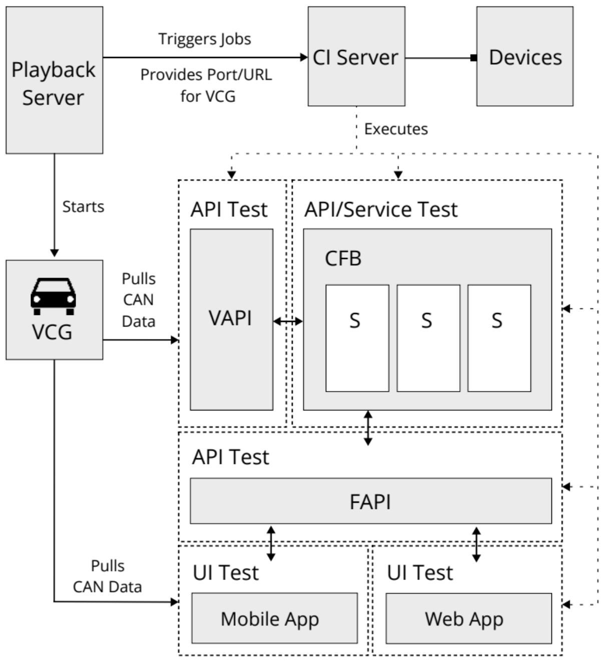

Figure 5. Deep Testing Architecture

B. CI Server

The CI Server will contain the test jobs running against the APIs, services and applications. Trigger can be either the Playback Server, a specific event or a time interval for the CI server to start the process. The Playback Server provides the CI server, i.e. the test jobs, with the connection information for the started VCG instance, consisting of URL and port. The test job uses the URL and port to instantiate an EXLAP client to establish the data stream to the VCG instance and pulls the CAN data. When the connection and stream is established, the actual test script within the test job will be executed. The services and APIs will be running as test instances on other servers. These need to be accessible for the test jobs. For early development and setup, mocked service instances can be useful.

C. Devices

Devices will be connected to the CI server or any available nodes. When connecting mobile devices the completion of all verification steps must be done to enable communication and debugging [19]. For testing purpose emulators can be used to speed up development of the infrastructure. Another alternative is a device farm or device cloud where test devices and hardware are provided as a service [25].

V. Running System

This section explains the methodology of initiating and operating the test system. For illustration, the process of an actual test case will be examined.

A. *Test Setup*

The example test case is kept simple. The user logs in, starts a car trip, records the logbook and ends the trip. We want to test the correctness of the VAPI data and perform an UI test on the android and web application. The test data used for the test case originates from a 30 minutes drive with a Volkswagen Passat GTE. The extracted logfile is placed within the VCG for playback.

On the CI Server a test job has to be created. The build jobs need to employ a build script that checks out the repository containing the test code and triggers the test script builds and executions. The CI Server must expose a REST API or other interface to enable communication to the Playback Server. The Playback Server will then use the REST interface to trigger test runs. The VCG will be running on a typical web service host.

B. *API & Service Test Script*

All API and service test scripts for VAPI, CFB and FAPI follow the same pattern of verifying incoming data and checking the output data. To illustrate the process the VAPI test is examined. The VAPI test script runs during the complete test run. The script checks if the data sent by the VCG and the mobile app are received and transformed by the VAPI as expected in conjunction with the test drive. As seen in Fig. 6 the data will be verified before and after the transformation. Due to the fact that the exact data transmitted by the test drive is known, its correctness can be verified.

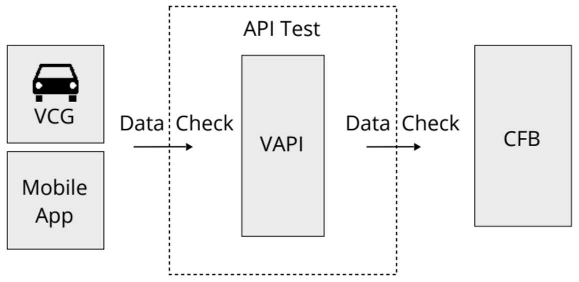

Figure 6. API Test

C. *Mobile Test Script*

The mobile UI test script is written in Java, utilizing the espresso framework for android. The mobile application is tested throughout the complete vehicle test drive. The test job will perform following steps.

D. *Web Test Script*

The UI test script for the web application uses the selenium framework. The web app will be tested after the test drive has been ended and recorded. The web app acts as an information portal, where all recorded trips to certain vehicles or drivers can be reviewed. Thus no live interaction occurs. These test steps need to performed.

VI. Experiments and Results

For the applicable use cases of the product ConnectedVan test drives have been recorded and linked to specific test cases. In our first steps we were able to generate test scripts for creating a driver's logbook, taking a business trip, creating a driver's entry and the edge case of an empty drive without data. The first use case was examined in detail in Section V and was run 1119 times in total on a CI Server.

632 runs were successful, 312 failed and 175 contained build error, as seen in Fig. 7. Build error means that the test job itself was faulty and contained mistakes, which happened in early development stages. For the 312 failed build several factor have to be considered. The product is still in development, therefore the frontends and APIs were target of major changes. Anytime that happened, the tests had to be adjusted as well. If not, the test failed.

In total 56 unique bugs were found by the infrastructure, which are not related to any API or frontend changes. The major part of the bugs consisted of frontend bugs, which included wrong wordings, missing fields or buttons and responsiveness. In terms of bugs concerning the APIs and backends, several bugs were located with missing data fields or unexpected values.

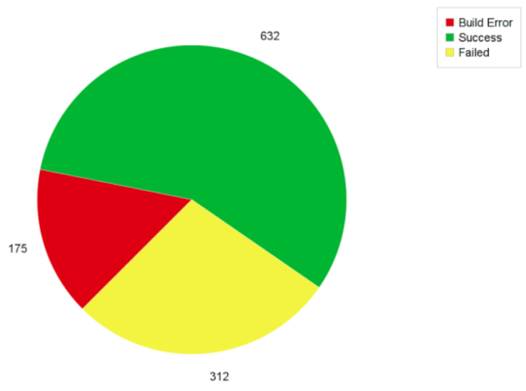

Figure 7. Test Statistics

VII. Conclusions

With the DeepTesting Architecture we were able to record and store test drives. We can launch them in an automated way in conjunction with the corresponding test cases triggered by new commits of the developers.

A. *Positive*

The architecture offers end-to-end testing through all components of our software system from test drive generation over APIs and services to the UI. We successfully integrated use cases to be tested by the architecture throughout all components. By splitting the architecture into smaller components like the VCG, Playback Server and the CI Server scaling becomes more manageable. Multiple VCG instances can run at any time, while having multiple test job instances running on the CI server or CI slave. The Playback Server only has to trigger and forward the information for the test jobs. For large scale UI testing however, a device farm or device cloud would be necessary.

B. Negative

The system needs a tremendous effort of developer operations to enable communication and data flow through all parts of the application. Communication must also guarantee stability and integrity. As various components might be added to the overall system, adjustments have to be made for the test automation flow. This outlines another effort which accompanies the architecture, the effort of maintenance. Maintenance is a critical factor in test automation for long term development. If adjustments at the architecture become more time consuming than the actual testing, the architecture becomes obsolete.

Another field of improvement, though not directly connected to the architecture itself is the generation of test data. If a large variety of test drives are to be available, the actual test drives have to be performed with a real car in the real world. For edge cases like car crashes or similar – safety, time and resources are predominant factors to be considered.

Executing a test run takes up the exact duration of the linked test drive logfile. This can cause complications when the whole system needs to be tested quickly, in case of time pressure. However there are current methods and technologies to fast forward the simulation process which will be implemented in the future.

C. Future

In the future more test cases will be examined and executed to improve the stability and integrity of the architecture. A possible improvement will be the usage of device farms to enable a larger scale of test runs. Also the impact of the architecture on the resulting code quality, refactoring effort and software stability will be analyzed, thus only being measurable over a longer time frame.

Furthermore a GUI will be created that enables non technical users to create test cases. Single fragments of tests can be put together to create a test suite.

ACKNOWLEDGMENT

The authors wish to thank Dr. Joost Garen, Dr. Timo Graen, Fabian Hafen, Timo Krallmann, Martin Doering and Kevin Schulze for the great support in the review and refining process of this paper.

REFERENCES

[1] G. Iyer, “Connected cars - a state of the industry report,” 2016.

[2] F. Holmes, K. Nolan, C. Schreiner, and C. Dodge, “Special report: Artificial intelligence and the auto industry,” 2017.s

[3] “Volkswagen ConnectedVan,” accessed: 2017-11-3. [Online]. Available: https://connectedvan.volkswagen-commercial-vehicles.com/jctcvanfleetmgrTMP/connectedvan/fleetmanager/login

[4] B. Fijalkowski, *Automotive Mechatronics: Operational and Practical Issues*, Springer Science, 2011, vol. 1.

[5] D. Hoffman, “Test automation architectures: planning for test automation,” in *Proc. Internation Quality Week*, 1999.

[6] S. T. und S. Sinha und Nimit Singhania and S. Chandra, “Automating Test Automation,” *ICSE*, 2012.

[7] “Test Automation Market by Test Type (Functional Testing, Configuration Testing, Web Services Testing, Acceptance Testing, Compatibility Testing, Integration Testing, Load Testing, Security Testing, Mobile Testing, Migration Testing, Platform Testing, Usability Testing, Network Testing and QA Process Design): Global Industry Perspective, Comprehensive Analysis, and Forecast, 2016 - 2022,” Tech. Rep., 2016.

[8] G. Myers, *The Art of Software Testing*. Wiley, 2011.

[9] R. R. und Wolfmaier K., “Economic perspectives in test automation: balancing automated and manual testing with opportunity cost,” in *Proc. the 2006 international workshop on Automation of software test*, p. 8591, 2006.

[10] T. Xie, “Towards a framework for different unit testing of object-oriented programs,” 2012.

[11] M. Fowler, “Mocks aren’t Stubs,” 2007.

[12] Y. W. Jerry Zeyu Gao, H. S. Jacob Tsao, *Testing and Quality Assurance for Component-based Software*, Artech House, 2003.

[13] L. Williams, “White-Box Testing,” pp. 60 – 61, 2013.

[14] A. Hunt and D. Thomas, *The Pragmatic Programmer*, Addison Wesley Longman, Inc., 2000.

[15] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999.

[16] “Efficient Regression Tests for Database Applications,” *Springer Journal*, 2006.

[17] F. Benoit, J. Ramachandran, T. Kaddoura, and G. Branco, “Automated testing on devices,” accessed: 2018-01-17. [Online]. Available: https://medium.com/netflix-techblog/automated-testing-on- devices-fc5a39f47e24

[18] M. Brück, “Buch-Profis übernehmen Thalia,” accessed: 2018-02-09. [Online]. Available: http://www.wiwo.de/unternehmen/handel/buchhandel-buch-profis- uebernehmen-thalia/13858616.html

[19] R. Steller, “Entwicklung einer Testautomatisierungs-Infrastruktur für mobile Applikationen,” B.S. thesis, Beuth University of Applied Science, Berlin, Germany, 2015.

[20] C. K. C. Mohammad Ali Darvish Darab, “Black-box test data generation for GUI testing,” 2014.

[21] A. Höpfner and T. Fuchs, “Hardwarespezifikationen Telematikbox CaTe,” Tech. Rep., 2015.

[22] O. Hartkopp, “Programmierschnittstellen für eingebettete Netzwerke in Mehrbenutzerbetriebssystemen am Beispiel des Controller Area Network,” pp. 215 – 225, 2009.

[23] J. Krüger, “VirtualCargate 2nd Edition,” *Tech. Rep*., 2012.

[24] J. Krüger and H. C. Fricke, “EXLAP Specification version 1.3,” Volkswagen AG, Tech. Rep., 2012.

[25] T. Aaltonen, V. Myllarniemi, M. Raatkainen, N. Makitalo, and P. Jari, An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud, ” *IEEE*, 2013.

Robin Steller was born in Berlin, Germany, in 1993. He acquired the B.S. degree and M.S. degree in media informatics from the Beuth University of Applied Science Berlin, Germany, in 2012 and 2018.

He started working as an Automation Engineer at Thalia GmbH in 2014, creating test automation architectures for mobile applications. In 2016 he became a Software Developer at the European IT Consultancy, Berlin, working on human resource tools. In 2017 he worked at Doozer GmbH, Berlin, as a Frontend Developer. In October 2017 he began working on his master thesis in the fields of automation at Volkswagen Commercial Vehicles, Hanover, until May 2018. He currently works as a Frontend Developer at Aperto – An IBM Company in Berlin. His current field of research includes automation, testing, cloud infrastructures and frontend architectures.

Robin Steller was a member of the Microsoft Student Partners and the appointments committee of data science at the Beuth University of Applied Science in Berlin, Germany.

Dr. Marek Stess Dr. Marek Stess was born in Katowtice, Poland in 1985. In 1990 his family and he went to Germany where he received the Bachelor of Science degree in computer science in 2009 from the Leibniz University Hanover. Directly after his B. Sc. he started the Master of Science study at the Leibniz University Hanover in computer science. He achieved the M. Sc. in 2013.

After the degree M. Sc. he started his Phd at the Volkswagen Group in the field of automated driving in cooperation with the Institute of Systems Engineering – Real Time Systems Group of the Leibniz University Hanover. He developed a localization algorithm for automated vehicles by using characteristics of the surrounding environment as different road markings and column shaped objects. He acquired the Phd in 2016 and went to Volkswagen Commercial Vehicles to work on industrial mobile online services.

Dr. Marek Stess is currently responsible for the Volkswagen Commercial Vehicles product Connect Fleet.